Volba programovacího jazyka pro implementaci šifer

Markéta Šustková

Bakalářská práce
2021

Univerzita Tomáše Bati ve Zlíně
Fakulta aplikované informatiky

Univerzita Tomáše Bati ve Zlíně
Fakulta aplikované informatiky
Ústav informatiky a umělé inteligence

Akademický rok: 2020/2021

ZADÁNÍ BAKALÁŘSKÉ PRÁCE

(projektu, uměleckého díla, uměleckého výkonu)

Jméno a příjmení:	**Markéta Šustková**
Osobní číslo:	**A18080**
Studijní program:	**B3902 Inženýrská informatika**
Studijní obor:	**Softwarové inženýrství**
Forma studia:	**Prezenční**
Téma práce:	**Volba programovacího jazyka pro implementaci šifer**
Téma práce anglicky:	**Choice of Programming Language for Cipher Implementation**

Zásady pro vypracování

1. Nastudujte a popište problematiku kryptologie od historické až po moderní.
2. Nastudujte a popište problematiku spojenou s optimalizací kódu a výkonnostními testy.
3. Zvolte 3 vhodné zástupce šifrovacích algoritmů a implementujte je pomocí 3 programovacích jazyků.
4. Vytvořte webovou stránku jako ukázku a podklad pro rozhodování při volbě programovacího jazyka v rámci předmětu Kryptologie.
5. Vhodně vyhodnoťte a reprezentujte výsledky.

Forma zpracování bakalářské práce: **Tištěná/elektronická**

Seznam doporučené literatury:

1. PIPER, F. C. a Sean MURPHY. Kryptografie. Praha: Dokořán, 2006. Průvodce pro každého. ISBN 80-736-3074-5.
2. SINGH, Simon. Kniha kódů a šifer: Utajování od starého Egypta po kvantovou kryptografii. Praha: Dokořán, 2003. Aliter (Argo: Dokořán): Dokořán). ISBN 80-86569-18-7.
3. BERLOQUIN, Pierre, 2011. Skryté kódy a velkolepé projekty: tajné jazyky od starověku po současnost. Praha: Knižní klub, s. 193-194. Universum (Knižní klub). ISBN 9788024228471.
4. JANEČEK, Jiří, 1994. Odhalená tajemství šifrovacích klíčů minulosti: ruční šifry. Praha: Naše vojsko. Mozaika (Naše vojsko). ISBN 80-206-0462-6.

Vedoucí bakalářské práce: **Ing. Petr Žáček**
Ústav informatiky a umělé inteligence

Datum zadání bakalářské práce: **15. ledna 2021**
Termín odevzdání bakalářské práce: **17. května 2021**

doc. Mgr. Milan Adámek, Ph.D. v.r.
děkan

UNIVERZITA TOMÁŠE BATI VE ZLÍNĚ
Fakulta aplikované informatiky
DĚKAN

prof. Mgr. Roman Jašek, Ph.D. v.r.
ředitel ústavu

Ve Zlíně dne 15. ledna 2021

Prohlašuji, že

- beru na vědomí, že odevzdáním bakalářské práce souhlasím se zveřejněním své práce podle zákona č. 111/1998 Sb. o vysokých školách a o změně a doplnění dalších zákonů (zákon o vysokých školách), ve znění pozdějších právních předpisů, bez ohledu na výsledek obhajoby;
- beru na vědomí, že bakalářská práce bude uložena v elektronické podobě v univerzitním informačním systému dostupná k prezenčnímu nahlédnutí, že jeden výtisk bakalářské práce bude uložen v příruční knihovně Fakulty aplikované informatiky Univerzity Tomáše Bati ve Zlíně;
- byl/a jsem seznámen/a s tím, že na moji bakalářskou práci se plně vztahuje zákon č. 121/2000 Sb. o právu autorském, o právech souvisejících s právem autorským a o změně některých zákonů (autorský zákon) ve znění pozdějších právních předpisů, zejm. § 35 odst. 3;
- beru na vědomí, že podle § 60 odst. 1 autorského zákona má UTB ve Zlíně právo na uzavření licenční smlouvy o užití školního díla v rozsahu § 12 odst. 4 autorského zákona;
- beru na vědomí, že podle § 60 odst. 2 a 3 autorského zákona mohu užít své dílo – bakalářskou práci nebo poskytnout licenci k jejímu využití jen připouští-li tak licenční smlouva uzavřená mezi mnou a Univerzitou Tomáše Bati ve Zlíně s tím, že vyrovnání případného přiměřeného příspěvku na úhradu nákladů, které byly Univerzitou Tomáše Bati ve Zlíně na vytvoření díla vynaloženy (až do jejich skutečné výše) bude rovněž předmětem této licenční smlouvy;
- beru na vědomí, že pokud bylo k vypracování bakalářské práce využito softwaru poskytnutého Univerzitou Tomáše Bati ve Zlíně nebo jinými subjekty pouze ke studijním a výzkumným účelům (tedy pouze k nekomerčnímu využití), nelze výsledky bakalářské práce využít ke komerčním účelům;
- beru na vědomí, že pokud je výstupem bakalářské práce jakýkoliv softwarový produkt, považují se za součást práce rovněž i zdrojové kódy, popř. soubory, ze kterých se projekt skládá. Neodevzdání této součásti může být důvodem k neobhájení práce.

Prohlašuji,

- že jsem na bakalářské práci pracoval samostatně a použitou literaturu jsem citoval. V případě publikace výsledků budu uveden jako spoluautor.
- že odevzdaná verze bakalářské práce a verze elektronická nahraná do IS/STAG jsou totožné.

Ve Zlíně, dne 5. května 2021

Markéta Šustková, v. r.
podpis studenta

ABSTRAKT

Tato bakalářská práce slouží jako základní přehled o problematice kryptografie, konkrétně od historických až po moderní algoritmy. Dále se práce zaměřuje na problematiku volby vhodného programovacího prostředí pro implementaci šifer. V rámci BP jsou vybrány tři šifry, které jsou svými vlastnostmi vhodnými zástupci v rámci kryptografie. Jsou zvoleny 3 programovací jazyky a šifry naimplementovány. Výsledné řešení je porovnáno z hlediska úsilí, které je nutné vynaložit na implementaci šifer a z hlediska výkonnosti. Vybrané šifry jsou naimplementovány i s graficky uživatelským rozhraním (GUI). Všechny postupy a poznatky (poznámky k implementaci) jsou vhodně vizualizovány pomocí webové stránky. Celkově práce bude sloužit jako „odrazový" můstek při volbě programovacího jazyka pro studenty předmětu Kryptologie a usnadní jim volbu jazyka pro vypracování úloh v rámci předmětu.

Klíčová slova: kryptografie, implementace, vývojové prostředí, programovací jazyk, Python, Java, C#

ABSTRACT

This bachelor thesis serves as a basic overview of cryptography, especifically from historical to modern algorithms. Furthermore, the thesis focuses on the issue of choosing a suitable software environment for the implementation of ciphers. Within BP, three ciphers are selected, which are characterized by their suitable representatives from cryptography. 3 programming languages are chosen and ciphers are implemented. Final solution is compared in terms of the work required to implement ciphers and in terms of their performance. Selected ciphers are also graphically implemented by the user interface (GUI). All procedures and findings (implementation notes) are visualized by using a web page. Overall, the work will serve as a steppingstone in choosing a programming language for students of the subject of cryptology and helps them to determine the choice of languages for the elaboration of tasks within the course.

Keywords: cryptography, implementation, Integrated development environment, programming language, Python, Java, C#

Ráda bych poděkovala mému vedoucímu bakalářské práce, panu Ing. Petru Žáčkovi, za cenné rady a trpělivost při konzultacích. Dále bych chtěla poděkovat za konzultace ohledně problematiky jazyka C# panu Ing. Vogeltanzovi. Další poděkování patří mé rodině a příteli, kteří mě po celou dobu studia plně podporují.

Prohlašuji, že odevzdaná verze bakalářské práce a verze elektronická nahraná do IS/STAG jsou totožné.

OBSAH

ÚVOD

Přemýšleli jste někdy nad tím, jak se posílaly zprávy například ve válkách, když je neměl přečíst nikdo jiný, než pověření lidé? Nebo jak probíhala komunikace v období králů a královen, kdy nebyla možnost žádných sociálních sítí, žádných ověřování identity pomocí elektronického podpisu či certifikátů? Dobu, kdy se nechodilo vybírat své vydělané peníze do zabezpečených krabiček, které nazýváme bankomaty, a už vůbec nebyla možnost zaplatit svůj nákup pouze pípnutím mobilu, který uchovává informace o vaší platební kartě a ověřujete platbu pouze svým otiskem prstu, či rysy obličeje? Dobu, kdy nebylo nutností, aby se mohly předávat zprávy bez možnosti přístupu 3. strany, nebo alespoň s omezeným přístupem. Období, kdy bylo využíváno jednoduché ruční šifrování.

Umění a věda utajování zpráv je označována jako kryptologie. Se vznikem schopnosti psaní a s vývojem lidské společnosti, s organizací lidstva do skupin, království a států se začala zvyšovat i potřeba skrývání informací před nepřáteli v boji o moc apod. Za její úplný počátek, kdy se začala objevovat, může být považováno používání hieroglyfů přibližně před 4000 lety Egypťany.

Když se pozorně rozhlédneme kolem nás, můžeme šifrování a kódy vidět na každém rohu. Ať už se jedná o ověřovací kódy k odemčení domu či auta nebo certifikáty a protokoly, které nám umožňují přihlášení do bankovnictví nebo jiných chráněných účtů. Postupem času došlo k rozluštění historických šifer. I tato oblast se neustále vyvíjí a v dnešní době používáme moderní šifrování, například využívání elektronického podpisu.

Tato bakalářská práce popisuje principy šifer od historických, jako je Caesarova šifra, po ty moderní, například v podobě elektronického podpisu. Seznamuje čtenáře se základními pojmy kryptografie a s jejím rozdělením. Pro lepší pochopení je u jednotlivých šifer pro popis principu využito tabulek a obrázků. Dále jsou popsány možnosti optimalizace kódů a výkonnostních testů.

Praktická část se věnuje implementaci tří šifer. Každá je napsána v jazycích Python, Java a C#. Jsou zde zhodnoceny výhody a nevýhody použití právě těchto jazyků. Na základě práce by měl být pro budoucí studenty usnadněn výběr programovacího jazyka v předmětu Kryptologie, což zejména pro začínající programátory, kteří se setkali například jen s jedním z nich, bývá poněkud záludná otázka výběru na začátku semestru.

I. TEORETICKÁ ČÁST

1 KRYPTOLOGIE

Kryptologie je vědní obor zabývající se šifrováním a dešifrováním zpráv. Dělí se na kryptografii, kryptoanalýzu a její nedílnou součástí je i steganografie. Používá se k zabezpečení komunikace, informací, dat a zpráv pomocí speciálních protokolů a algoritmů. Cílem je zabránit dalším neoprávněným stranám ve čtení a porozumění přenášených informací.

Kryptografie se zabývá šifrováním zpráv – převedením zprávy do nečitelné podoby. Naopak kryptoanalýza řeší jejich prolamování a získání informací ze zašifrovaných dat i bez znalosti klíče. Kryptoanalýza se zaměřuje na bezpečnostní analýzu stávajících kryptosystémů.[1] Testuje, jak je daný algoritmus silný a odolný vůči prolomení. Steganografie není zašifrovaní zprávy jako takové. Jedná se o skrytí informace například do obrázku nebo audionahrávky. Snaží se informaci skrýt tak, aby nebyla na první pohled viditelná a nebyla odhalena. Často dochází k použití kombinace kryptografie společně se steganografií, kdy jsou data nejprve šifrována a pak skryta před neoprávněným přístupem k informaci. [1]

[1] sada algoritmů a protokolů pro šifrování, dešifrování a generování potřebných klíčů

2 KRYPTOGRAFIE

Kryptografie má za úkol šifrovat data. To znamená, že se z původní čitelné zprávy pomocí technik odvozených z matematických konceptů a souborů výpočtů založených na pravidlech nazývaných algoritmy stane shluk znaků, které na první pohled nedávají žádný smysl. Nemá za úkol data schovávat, pouze zabraňuje v jejich čitelnosti. Šifrování se používá zejména tehdy, když požadujeme chránit data například při posílání přes internet nebo si ukládáme informace do souborů a chceme zabránit v možnosti přečtení nepovolané osobě. S kryptografií se setkáme například i při prohlížení webu či transakcích kreditní kartou nebo přihlašováním do e-mailu.

2.1 Základní pojmy kryptografie

K pochopení principů jednotlivých šifer je potřeba znát základní terminologii, která se zde užívá. K základním pojmům patří:

- **Otevřený text** je původní zpráva, kterou následně chceme zašifrovat, aby bylo znemožněno její přečtení bez znalosti algoritmu a klíče.
- **Šifrový text** je již upravená zpráva, která prošla šifrovacím algoritmem a je zamezeno její přečtení nežádoucími stranami.
- **Šifrovací algoritmus** je souhrn předpisů aplikovaných na šifrování původního textu zprávy.
- V průběhu ochrany otevřeného textu dochází k **šifrování.** Šifrování je tedy převedení otevřeného textu do zašifrovaného, nečitelného.
- **Dešifrování** je opakem šifrování. Z nesmyslného shluku znaků – šifrového textu získáme původní zprávu, kterou jsme schopni přečíst.
- Prostřednictvím **šifrovacího klíče** dochází k šifrování. Klíč může být například řetězec různých znaků, který spolu s otevřeným textem tvoří vstupní parametry šifrovacího algoritmu. Měli bychom velmi apelovat na silný a bezpečný klíč, jinak může dojít k nežádanému prolomení šifry. Klíč by měl být známý pouze stranám, které si mezi sebou informaci předávají a jsou oprávněny jí porozumět.
- Klíč může být stejný pro šifrování i dešifrovaní. V tom případě se jedná o **symetrické šifrování**. Pokud hovoříme o **asymetrickém šifrování**, klíč obsahuje dvě části – veřejnou a soukromou. Pomocí veřejného klíče dochází většinou k šifrování a soukromý dešifrovací klíč naopak zprávu vrací do původního čitelného stavu. [2]

Ukázka principu šifrovacích algoritmů, kdy je na vstup přiveden otevřený text spolu s klíčem, z nějž vystupuje šifrový text, a po vstupu šifrového textu spolu s dešifrovacím klíčem do dešifrovacího algoritmu dostáváme na výstupu zpět původní otevřený text, je znázorněna na Obrázek 1.

Obrázek 1. Ukázka principu algoritmů, jejich vstupů a výstupů [1]

Kryptografie zahrnuje techniky jako jsou mikrotečky[2], slučování slov s obrázky, přehazování písmen v textu, záměna písmen za jiné a další. V dnešním počítačovém světě se kryptografie nejčastěji spojuje s šifrováním prostého textu a dešifrováním textu šifrového.

Mezi cíle kryptografie patří:

- Důvěrnost (bezpečnost) – informaci nemůže pochopit ten, pro koho nebyla zamýšlena
- Integrita dat – informace nemohou být změněny při skladování nebo přenosu mezi odesílatelem a příjemcem, aniž by byla zaznamenána změna. Nemůže dojít k nezaznamenanému mazání, přidávání nebo substituci dat. Toho lze dosáhnout hashováním
- Neodvolatelnost – nemělo by být možné popřít autorství zprávy
- Autentizace (identifikace) – odesílatel a příjemce si mohou navzájem potvrdit identitu. Autentizace může probíhat pomocí hesla, biometrických údajů nebo např. informací o kreditní kartě, kterou daná osoba vlastní

Kryptosystémy používají sadu postupů známých jako algoritmy nebo šifry pro šifrování a dešifrování zpráv pro zabezpečení komunikace. Sada šifry používá jeden algoritmus pro

[2] Technika používána zejména během 2. světové války. Kombinace se steganografií. Stránka textu je zmenšena do velikosti tečky, která je použita jako obyčejná tečka za větou v dopise. [3]

šifrování, jiný pro ověřování zpráv a další pro tvorbu klíčů. Tento proces zabudovaný do protokolů a napsaný v softwaru, který běží na operačních systémech, zahrnuje generování veřejného a soukromého klíče pro šifrování a dešifrování dat, digitální podepisování a ověřování pro autentizaci zpráv a výměnu klíčů. [2][4]

Kryptografii lze rozdělit na klasickou a moderní. Klasická kryptografie využívá zejména metod substituce a transpozice. Moderní šifry lze rozdělit podle klíčů, které jsou využívány, na symetrickou klíčovou kryptografii, hashovací funkce a asymetrickou klíčovou kryptografii.

Obrázek 2. Skytale – šifra využívána zejména Sparťany v období válek [6]

3 KLASICKÁ KRYPTOGRAFIE

Klasická šifra je typem šifry, která byla uplatňována spíše historicky. Nyní se již většinou nepoužívá. Toto odvětví šifer lze obvykle vypočítat a vyřešit ručně. Nicméně jdou také snadno prolomit moderní technologií. Klasická kryptografie zahrnuje jednoduché systémy používané od řeckých a římských dob jako šifra Caesarova, která patří mezi první zdokumentovanou substituční šifru používanou za účelem utajeně komunikovat během válek. Dále se zde řadí propracované renesanční šifry, zavádění opatření proti jednoduchému rozluštění šifer v podobě klamačů (znaků, které nemají v textu žádný význam – pouze matou nepřítele), kryptografie z druhé světové války jako stroj Enigma a další.

Taktéž v arabských zemích bylo šifrování pomocí substituce na denním pořádku. Stát fungoval díky účinné správě. Zdejší vykonavatelé úřednických funkcí potřebovali bezpečnost v komunikaci. Šifry byly používány jak na ochranu státních věcí, tak i na daňové záznamy. Pochází odtud i kniha zmiňující šifrování, která byla vydána v 10. století. Docházelo k aplikaci šifrové abecedy, kdy pouze přehodili písmena z abecedy otevřeného textu. Později používají již substituci, kdy jsou písmena z původní abecedy nahrazena různými znaky. Například písmeno A by se mohlo zaměnit za %, písmeno B za * a tak by se pokračovalo. Jedná se o mono-alfabetickou substituci (každé písmeno z otevřeného textu je nahrazeno pokaždé tím samým písmenem z šifrové abecedy). Došlo zde ke zlomu v šifrování. Zkoumáním četnosti slov ve zjeveních popsaných v Koránu a následně písmen, objevili techniku užívanou následně v kryptoanalýze k prolomení šifer, frekvenční analýzu. [3]

Klasické šifry lze rozdělit podle techniky, kterou využívají na substituční a transpoziční šifry. Hlavním rozdílem je, že při substituci jsou písmena otevřeného textu nahrazena jinými písmeny, čísly nebo symboly. Abeceda otevřeného textu je nahrazena jinou abecedou. Oproti tomu transpozice písmena nenahrazuje, abeceda otevřeného textu zůstává beze změny. Písmena otevřeného textu zůstávají stejná, pouze se uspořádají v jiném pořadí. K substitučním šifrám se řadí šifra mono-alfabetická, poly-alfabetická nebo šifra Playfair. Transpoziční technika užívá dvou forem, a to klíčovou nebo bezklíčovou transpoziční šifru. Cílem substituční techniky je změnit identitu entity, zatímco transpoziční technika mění pozici entity. Substituci lze rozeznat při frekvenční analýze, což je statická analýza četnosti znaků. Z frekvenční analýzy lze taky rozeznat, zda se jedná o text, na který byla použita metoda substituční či transpoziční. U transpozice zůstává frekvenční analýza původního a

zašifrovaného textu neměnná. Při transpozici musíme obvykle znát klíč, abychom dešifrovali text. Substituční šifry jsou jednodušší na prolomení, takže méně bezpečné. [5]

3.1 Substituční šifry

Mezi substituční šifry patří šifry mono-alfabetické jako Caesarova šifra, ATBASH, náhodná abeceda či lineární posun, poly-alfabetické jako Vigenérova šifra, dále homofonní, polygrafické, do kterých řadíme šifru Playfair, Bifid nebo Hillovu šifru a další.

Jak již bylo uváděno výše, substituce nahrazuje abecedu původního textu abecedou novou. Například slovo AHOJ. Při šifrování textu je určena místo původní abecedy A-Z abeceda nová, která je v tomto případě oproti klasické abecedě posunuta o jeden znak doprava. Písmeno A je tedy nahrazeno písmenem B, B nahrazuje znak C, C je v šifrovaném textu zapsáno jako D. Tak se pokračuje dále až po Z, které je nahrazeno znakem A. Ze vstupního slova AHOJ tedy vystupuje šifrový text BIPK.

3.1.1 Caesarova šifra

Caesar využíval k domluvě v průběhu válek mnoho dalších tajných způsobů, které sepsal Valerius Probus. Bohužel se žádné zápisky přímo o ostatních šifrách nedochovaly. Vůbec první zaznamenaná substituční šifra využita právě Caesarem byla ta, ve které nahradil římské znaky řeckými, které byly pro nepřítele nečitelné, a zprávu tak mohl bezpečně poslat. Tento počin je popsán v Zápiscích o válce galské od Julia Caesara. Asi jeho nejznámější šifra, která se v kryptologii využívá je šifra s pevným posunem znaků v abecedě. Jedná se o velmi jednoduchou šifru. Dnes je již snadno rozluštitelná díky frekvenční analýze. Dochází zde k posunu otevřeného textu o 3 místa v běžné abecedě. [3]

Otevřená abeceda	a b c d e f g h i j k l m n o p q r s t u v w x y z
Šifrová abeceda	D E F G H I J K L M N O P Q R S T U V W X Y Z A B C
Otevřený text	v e n i, v i d i, v i c i
Šifrový text	Y H Q L, Y L G L, Y L F L

Obrázek 3. Ukázka Caesarovy šifry [3]

Na prvním řádku obrázku Obrázek 3 znázorňujícím použití Caesarovy šifry je vypsána klasická abeceda textu před šifrováním. O řádek níže je vypsána abeceda, která je využívána k překladu původního textu do textu zašifrovaného. Postup šifrování textu je takový, že dojde k vyhledání každého písmene v horním řádku otevřené abecedy a toto písmeno je nahrazeno znakem, který mu náleží v abecedě na řádku druhém. Znaky, jenž nahrazují původní

písmena, se zapisují jako šifrový text. V příkladu je písmenu V, které si je možné vyhledat na prvním řádku, přiděleno písmeno Y z řádku druhého.

K šifrování lze využít také vzorec: $Ci = (Ti + k)\ mod\ m$, kde Ci je znak šifrovaného textu, Ti je index znaku textu původního, k je daný posun (v tomto případě 3) a za m se dosazuje délka abecedy. Například při šifrovaní písmene B s použitím anglické abecedy, jenž obsahuje 26 prvků, a se zvoleným posunem o 3 znaky, se dostává: $Ci = (1 + 3)\ mod\ 26 = 4 = E$. [7]

3.1.2 ATBASH

ATBASH je další známá mono-alfabetická šifra, která je považována za šifru nejstarší. Původně byla používána pro hebrejskou abecedu. Lze ji velmi snadno dešifrovat, protože neobsahuje žádný klíč. Používá pouze reverzi dané abecedy. První písmeno abecedy otevřeného textu je nahrazeno posledním, druhé předposledním a tak dále. Ukázka abeced ATBASH je znázorněna na obrázku Obrázek 4.

Plaintext Alphabet	a	b	c	d	e	f	g	h	i	j	k	l	m	n	o	p	q	r	s	t	u	v	w	x	y	z
Ciphertext Alphabet	Z	Y	X	W	V	U	T	S	R	Q	P	O	N	M	L	K	J	I	H	G	F	E	D	C	B	A

Obrázek 4. Abeceda šifry ATBASH [8]

Při šifrování jsou opět pouze nahrazeny písmena z otevřeného textu příslušnými znaky z abecedy textu šifrového. V tomto případě je slovo AHOJ zašifrováno do znaků ZSLQ. Dešifrování znaků probíhá téměř totožně jako šifrování. Znaky z šifrového textu jsou substituovány za písmena abecedy původního textu. Znak Z je nahrazen písmenem A, S je zaměněno za H, a postupuje se podobně dále.

Tento typ šifry může být zkomplikován tím, že budou do abecedy otevřeného textu přidány čísla 0 až 9, případně speciální znaky. Znovu se jedná o již historickou šifru. [9]

3.1.3 Šifra s klíčovým slovem

Další substituční šifrou je šifra s klíčovým slovem. U předchozích šifer nebylo potřeba žádného šifrovacího klíče. Šifrová abeceda se tvořila pouze posunem znaků abecedy původní. Nyní již uživatel musí zvolit jako klíč slovo, které je použito při tvorbě šifrové abecedy. Písmena v klíči musí být unikátní, takže po zvolení slova ještě dojde k odstranění duplicitních znaků. Následně je pomocí klíče vygenerována šifrová abeceda, kdy na její začátek je dáno slovo klíčové, a poté je postupně doplněna znaky z původní abecedy. Znaky, které již jsou obsaženy v klíčovém slově se vynechají. Každé písmeno je v abecedě pouze jednou.

Jako příklad poslouží zvolené klíčové slovo KRYPTOLOGIE. Musí být odstraněny duplicitní znaky, tedy klíč vypadá následovně: KRYPTOLGIE (odstraněno druhé písmeno O).

Původní anglická abeceda s písmeny A-Z je nahrazena abecedou: KRYPTOLGIEABCDFHJMNQSUVWXZ. Při šifrování je znak A nahrazen písmenem K, znak B písmenem R, C znakem Y atd. Například slovo AHOJ je zašifrováno do znaků KGFE. [10]

3.1.4 Užívání klamačů a nomenklátorů

Jelikož šifrování pomocí typu mono-alfabetické substituce díky objevu frekvenční analýzy nebylo nejbezpečnější, nastávala otázka, jak jej ztížit a více zabezpečit proti útokům kryptoanalytiků. Začali se užívat klamače. Do textu bylo jednoduše vsunuto pár bezvýznamných znaků či čísel. Osoby, kterým byl text určen, dešifrování nečinilo problém. Jednoduše tyto znaky přehlíželi. Účelem bylo navodit komplikace při analýze četnosti znaků případné odchycené komunikace a zmatení osob, kterým informace nebyly určeny.

Dále docházelo ke kombinaci šifrovaní každého písmene zvlášť a změně pouze některých slov za určité symboly nebo čísla takzvané zavedení nomenklátorů. Například místo slova král se při psaní použil znak *, slovo dnes bylo nahrazeno číslem 99 a tak dále. Jaké slova se nahrazují bylo jen na účastnících komunikace. Nomenklátory hojně využívala i známá panovnice Marie Stuartovna. S postupem času si však kryptoanalytici věděli rady i se složitějšími substitucemi, a i toto šifrování bylo prolomeno. Nevýhodou bylo zejména to, že se zde stále jednalo o jednoduchou substituci a ostatní znaky (zakódovaná slova) se dali odvodit z kontextu zprávy.

3.1.5 Homofonní substituce

Homofonní substituce patří stále mezi mono-alfabetické šifry. Vyznačuje se nahrazením každého písmene z abecedy OT několika odlišnými symboly. Jeden znak z OT může být nahrazen několika možnostmi, ale každý znak z abecedy ŠT značí pouze jedno písmeno z OT. Počet variant, jakými se v dané komunikaci jednotlivé znaky nahrazují, obvykle odpovídá relativní četnosti znaků v textu při užívání dané abecedy. Jestliže je písmeno zastoupeno ze 4 %, máme 4 varianty náhrady.

Příklad užití homofonní substituce je znázorněn pomocí tabulky Tabulka 1. Jsou zvoleny možnosti znaků, kterými lze nahradit písmena OT. V tomto případě slovo AHOJ může být

zašifrováno jako 55111352, ale i jako 90111873. Tímto se analýza a možnost proniknutí k původnímu textu bez oprávnění opět poněkud zkomplikovala.

Tabulka 1. Příklad homofonní šifry u slova AHOJ

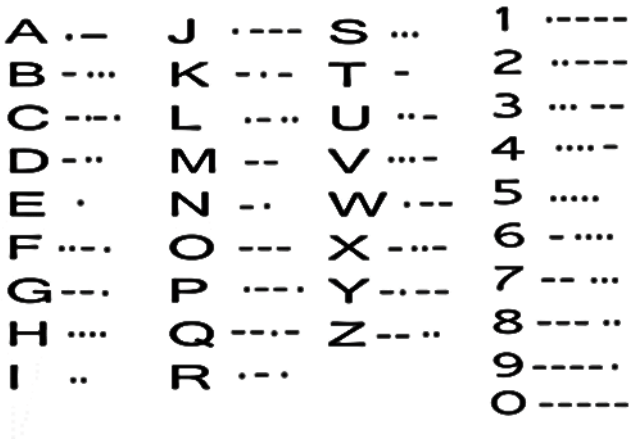
A: 01,55,79,02,09,90	H: 11	O: 12,13,14,15,16,17,18,88
J: 52,73		

3.1.6 Vigenérova šifra

Vývoj šifer dále směřoval k poly-alfabetickým šifrám. Znaky z původního textu jsou nahrazovány pomocí více šifrovacích abeced, ne pouze jedné. V 15. století byl nápad využití více šifrových abeced popsán Leonem Battistem Albertim, florentskou osobností renesance. Právě on navrhl šifrovat pomocí více šifrovacích abeced, kdy se první písmeno šifruje pomocí první, druhé pomocí druhé, třetí znak využívá opět první abecedu a tak dále. Alberti svoji myšlenku však již dále nerozvinul. Díky tomu mohl být tímto nápadem inspirován Blaise de Vigenére, po kterém je pojmenována právě Vigenérova šifra, jedna z poly-alfabetických šifer. Vigenére byl francouzský diplomat, jenž žil v 16. století. Společnosti se ale zdála poněkud složitá a plného využití našla až o 2 století později, kdy se kryptoanalýza stala opět důmyslnější a mono-alfabetické šifrování se stalo zaostalým.

V 18. století začaly fungovat tzv. černé komnaty. Centra, kde docházelo k hromadění informací a dešifrování zpráv. Pracovali zde i odborníci na jazyky, aby jazyková bariéra činila při luštění ten nejmenší problém. Například ve Vídni byly stahovány dopisy z pošty, z nichž se vytvořily kopie, které byly následně předány kryptoanalytikům. Dopis byl poté znovu zapečetěn a předán zpět na poštu, aby mohl být včas doručen. Jelikož docházelo k takto organizovanému luštění šifer, krypto-grafové byli donuceni využít a ocenit potenciál Vigenérovy šifry.

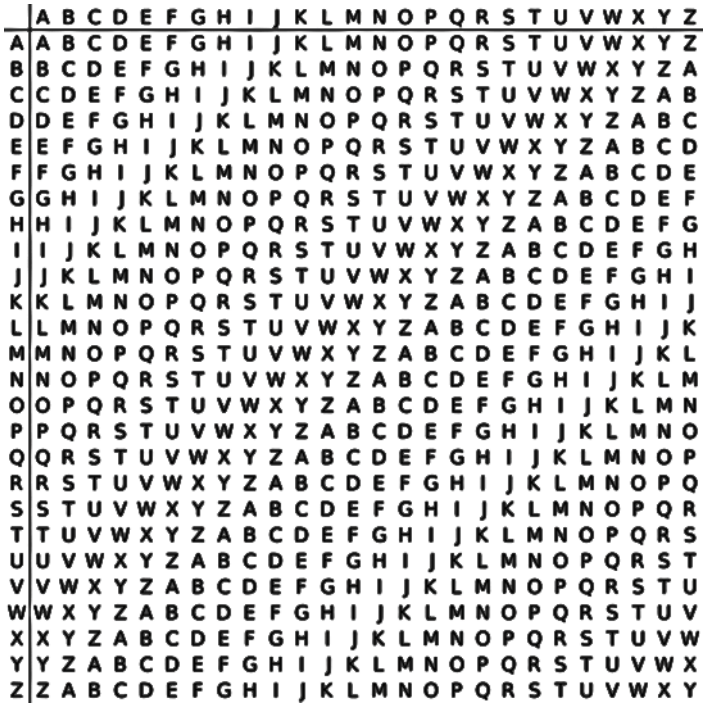
K použití Vigenérovy šifry také přispěla nutnost zabezpečit zprávy posílané pomocí telegrafu, jehož počátky sahají do této doby. Docházelo k používání Morseovy abecedy (znázorněna na obrázku Obrázek 5), která však texty nijak nešifrovala. Text byl šifrován ještě před zasláním telegrafem, a to právě prostřednictvím šifry pana Vigenéra, kde je pro šifrování jednoho písmena několik možností odpovídajících délce klíčového slova. Do Morseova kódu byl přeložen text již zašifrovaný a díky tomu pouze při přeložení z morseovky do běžného textu nebyl srozumitelný. [2][3]



Obrázek 5. Morseova abeceda [11]

Z počátku byla tato šifra brána jako nerozluštitelná. Že tomu tak není dokázal v 19. století Charles Babbage, známý svým navržením moderního počítače. Zkoumal více oblastí, od přírodních věd a letokruhů na stromě v souvislosti s klimatickými podmínkami v minulosti, až po informační technologie. Pro zajímavost navrhl například i jednotné poštovné, které se používá dodnes. Luštění šifer ho zaujalo již v dětství. Zaznamenal, že při použití šifry docházelo v ŠT k opakování shluků písmen. Byla zde tedy pravděpodobnost, že opakované shluky byly šifrovány stejnou částí klíče. To mu poskytlo základ pro prolomení šifry. Dokázal možnost frekvenční analýzy i u této poly-alfabetické šifry. Slabina šifry je v periodickém opakování klíčového slova a v používaní skupin písmen v daném jazyce. [3][13]

Vigenérova šifra využívá tabulky znázorněné na obrázku Obrázek 6. Horní řádek představuje písmena OT. Sloupec nalevo udává písmena klíče. V tabulce je obsaženo 26 monoalfabetických substitucí. Volí se klíč. Pokud klíč není stejně dlouhý jako původní text, neustále se opakuje. Při šifrování se nachází průnik písmen z OT a klíče, kdy výsledkem je zašifrované písmeno. Například pro slovo AHOJ se zvoleným heslem KLIC bude postup následující. Nejprve se nalezne průnik písmene A z horního řádku s písmenem K z levého sloupce. Výsledkem průniku je písmeno K, což je hledaný šifrový znak. Z písmene H za použití klíčového L vychází šifrové písmeno S. Takový postup se provádí až do konce otevřeného textu. Z OT AHOJ vychází šifrované slovo KSWL. Pro přehlednost jsou OT spolu s klíčem a ŠT uvedeny v tabulce Tabulka 2. [2]



	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O	P	Q	R	S	T	U	V	W	X	Y	Z
A	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O	P	Q	R	S	T	U	V	W	X	Y	Z
B	B	C	D	E	F	G	H	I	J	K	L	M	N	O	P	Q	R	S	T	U	V	W	X	Y	Z	A
C	C	D	E	F	G	H	I	J	K	L	M	N	O	P	Q	R	S	T	U	V	W	X	Y	Z	A	B
D	D	E	F	G	H	I	J	K	L	M	N	O	P	Q	R	S	T	U	V	W	X	Y	Z	A	B	C
E	E	F	G	H	I	J	K	L	M	N	O	P	Q	R	S	T	U	V	W	X	Y	Z	A	B	C	D
F	F	G	H	I	J	K	L	M	N	O	P	Q	R	S	T	U	V	W	X	Y	Z	A	B	C	D	E
G	G	H	I	J	K	L	M	N	O	P	Q	R	S	T	U	V	W	X	Y	Z	A	B	C	D	E	F
H	H	I	J	K	L	M	N	O	P	Q	R	S	T	U	V	W	X	Y	Z	A	B	C	D	E	F	G
I	I	J	K	L	M	N	O	P	Q	R	S	T	U	V	W	X	Y	Z	A	B	C	D	E	F	G	H
J	J	K	L	M	N	O	P	Q	R	S	T	U	V	W	X	Y	Z	A	B	C	D	E	F	G	H	I
K	K	L	M	N	O	P	Q	R	S	T	U	V	W	X	Y	Z	A	B	C	D	E	F	G	H	I	J
L	L	M	N	O	P	Q	R	S	T	U	V	W	X	Y	Z	A	B	C	D	E	F	G	H	I	J	K
M	M	N	O	P	Q	R	S	T	U	V	W	X	Y	Z	A	B	C	D	E	F	G	H	I	J	K	L
N	N	O	P	Q	R	S	T	U	V	W	X	Y	Z	A	B	C	D	E	F	G	H	I	J	K	L	M
O	O	P	Q	R	S	T	U	V	W	X	Y	Z	A	B	C	D	E	F	G	H	I	J	K	L	M	N
P	P	Q	R	S	T	U	V	W	X	Y	Z	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O
Q	Q	R	S	T	U	V	W	X	Y	Z	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O	P
R	R	S	T	U	V	W	X	Y	Z	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O	P	Q
S	S	T	U	V	W	X	Y	Z	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O	P	Q	R
T	T	U	V	W	X	Y	Z	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O	P	Q	R	S
U	U	V	W	X	Y	Z	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O	P	Q	R	S	T
V	V	W	X	Y	Z	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O	P	Q	R	S	T	U
W	W	X	Y	Z	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O	P	Q	R	S	T	U	V
X	X	Y	Z	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O	P	Q	R	S	T	U	V	W
Y	Y	Z	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O	P	Q	R	S	T	U	V	W	X
Z	Z	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O	P	Q	R	S	T	U	V	W	X	Y

Obrázek 6. Vigenérův čtverec [12]

Tabulka 2. Přehled zašifrování slova AHOJ Vigenérovou šifrou

OT	A	H	O	J
klíč	K	L	I	C
ŠT	K	S	W	L

3.1.7 Playfair

Playfair šifra patří do skupiny šifer polygrafických, které rozdělují původní text na skupinky písmen a ty jsou následně šifrovány předdefinovanými znaky.

Způsob šifrování Playfair šifrou byl vynalezen v 19. století. Na vývoji šifry spolupracovali sir Charles Wheatstone a baron Lyon Playfair. Byla užívána například v búrské válce, či ve válkách světových. Jedná se o bigramovou šifru. To znamená, že jsou šifrovány dvojice písmen OT, nikoli pouze jednotlivá písmena, jak tomu bylo dříve.

Klíčem je tabulka o rozměru 5 x 5 políček. Jelikož má tabulka pouze 25 znaků, musí dojít k nahrazení jednoho znaku z klasické abecedy obsahující znaků 26. Například písmeno J je v textu nahrazeno písmenem I. Postup šifrování začíná postupným rozdělením původního textu po dvojicích znaků. Jakmile se ve dvojici ocitnou dvě stejná písmena, vloží se mezi ně například písmeno X. Pokud je počet všech znaků lichý, vloží se další znak X na konec textu.

Je zvolen klíč, který je bez duplicitních znaků zapsán do tabulky 5 x 5 a tabulka je následně doplněna danou abecedou. Písmena, jež jsou v tabulce již obsažena, se při doplnění vynechají. Pro ukázku je jako původní zpráva zvoleno slovo AHOJ. Jelikož má tabulka jen 25 možností znaků, písmeno J je nahrazeno znakem I. Slovo na šifrování bude tedy AHOI. Jako klíč je vybráno seskupení znaků KZICEK. V klíčové tabulce bude klíč společně s abecedou zapsán tak, jak je uvedeno v tabulce Tabulka 3.

Tabulka 3. Ukázka klíčové tabulky Playfair šifry

K	Z	I	C	E
A	B	D	F	G
H	L	M	N	O
P	Q	R	S	T
U	V	W	X	Y

Při zapisování klíče KZICEK do tabulky se v tomto případě vynechá druhé písmeno K, aby byly odstraněny duplicitní znaky. Dále je zpráva AHOI rozdělena do dvojic znaků. Je zkontrolováno, zda zpráva nemá lichý počet znaků nebo zda dvojice neobsahují stejná písmena. Ve slově AHOI jsou splněny všechny požadované vlastnosti, tudíž k textu nemusí být přidáváno žádné písmeno X. Rozdělené slovo vypadá: AH OI.

Pro šifrování zde platí 3 pravidla:

- Pokud jsou obě písmena z právě šifrované dvojice na stejném řádku, každé z nich je nahrazeno písmenem, které se od něho nachází v tabulce o jednu pozici vpravo.
- V případě, že jsou písmena šifrované dvojice ze stejného sloupce, je každé nahrazeno písmenem o jednu pozici níže.
- Jestliže předchozí pravidla neplatí a písmena ze dvojice jsou v klíčové tabulce umístěna v různých řádcích a sloupcích, platí pravidlo následující. Písmena jsou postupně nahrazena znakem, který leží v průniku řádku písmene, které je nahrazováno, a sloupce druhého písmene ze dvojice.

Při postupu šifrování podle pravidel uvedených výše se dvojice písmen AH šifruje na HP (AH se nachází ve stejném sloupci → každé písmeno je nahrazeno znakem pod ním) a druhá dvojice OI odpovídá znakům ME (písmena OI se nachází v různých řádcích i sloupcích →

jsou nahrazena podle 3. pravidla pomocí průniku). Pro názornost jsou znaky označeny v tabulce Tabulka 3. Dešifrování probíhá inverzním způsobem. [2][14]

3.1.8 Bifid šifra

Šifra Bifid byla vynalezena Felixem Decastelle z Francie v 19. století. Bifid byla používána pouze k amatérskému šifrování. Využívá podobně jako šifra Playfair klíčovou tabulku o rozměru 5 x 5 polí. Z důvodu možnosti pouze 25 znaků v tabulce jsou opět písmena J nahrazována znakem I. Při šifrování jsou pod každé písmeno z OT zapsána 2 čísla, jedno znázorňuje číslo řádku a druhé číslo sloupce. Číslo řádku, na kterém se dané písmeno nachází je zapsáno do řádku horního a číslo sloupce se zapisuje pod něj. Jakmile jsou čísla zapsána u veškerých písmen OT, jsou rozdělena do bloků, například po pěti zleva doprava. Jsou rozdělovány zvlášť řádky a zvlášť sloupce. Dále jsou v jednotlivých blocích sloučeny řádky a po dvojicích čísel jsou určována zašifrovaná písmena. Dvojice čísel nyní uvádějí souřadnice řádku a sloupce písmene po zašifrování v klíčové tabulce. [15][16]

Tabulka 4. Klíčová tabulka šifry Bifid

	1	2	3	4	5
1	A	B	E	C	D
2	F	G	H	I	K
3	L	M	N	O	P
4	Q	R	S	T	U
5	V	W	X	Y	Z

Jako příklad je zde uvedena klíčová tabulka Tabulka 4. Za klíč je zvoleno slovo ABECEDA. Před zapsáním do tabulky jsou odstraněny duplicitní znaky. Znaky s odpovídajícími čísly z postupu šifrování věty AHOJ JAK SE MAS jsou vypsány v tabulce Tabulka 5, kdy je v textu samozřejmě nahrazeno písmeno J za znak I.

Tabulka 5. Vypsaná čísla při šifrování Bifid

A	H	O	I		I	A	K		S	E		M	A	S
1	2	3	2		2	1	2		4	1		3	1	4
1	3	4	4		4	1	5		3	3		2	1	3

Dále jsou čísla spojena do bloků po 5, což je znázorněno v tabulce Tabulka 6.

Tabulka 6. Čísla v blocích po pěti – Bifid šifra

1	2	3	2	2		1	2	4	1	3		1	4
1	3	4	4	4		1	5	3	3	2		1	3

Dále se pomocí dvojic čísel, které se berou po blocích zvlášť, a to nejprve první řádek daného bloku a poté řádek druhý, šifruje původní zpráva podle klíčové tabulky (dvojice určují souřadnice písmen, jak bylo zmíněno výše). Dvojice čísel spolu s již zašifrovaným textem jsou zobrazeny v tabulce Tabulka 7. Zpráva již zašifrovaná Bifidem zní FHBST FCEPH QL. Dešifrování se provádí opačným postupem.

Tabulka 7. Dvojice čísel z bloků a odpovídající šifrový text – Bifid

12	32	21	34	44		12	41	31	53	32		14	13
F	H	B	S	T		F	C	E	P	H		Q	L

Šifra Trifid je velmi podobná jako šifra Bifid. Rozdílem je používání trojrozměrné tabulky (3 x 3 x 3) místo tabulky 5 x 5.

3.1.9 Hillova šifra

Hillova šifra byla vyvinuta v první polovině 20. století a stejně jako šifra Playfair se řadí mezi substituční polygrafické šifry. Využívá znalosti matematické teorie čísel. Dochází k použití matic a jejich násobení. Pro společnost se zdála být velmi složitá, tak Lester S. Hill, vynálezce šifry, postavil systém šifry z ozubených koleček a řetězců. Toto však nikdy nenalezlo úspěch.

Při šifrování Hillovou šifrou se znaky abecedy označí od 0 po číslo 25, jak je zobrazeno v tabulce Tabulka 8. Číslo 0 představuje písmeno A a číslo 25 potom písmeno Z.

Tabulka 8. Abeceda převedená na čísla – Hillova šifra

A	B	C	D	E	F	G	H	I	J	K	L	M	N	O	P	Q	R	S	T	U	V	W	X	Y	Z
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	18	20	21	22	23	24	25

Jako klíč je zvolena matice o stejném počtu řádků i sloupců. Matice se skládá z klíčového slova, které musí být převedeno na čísla podle tabulky Tabulka 8. Takto převedená matice je viditelná v rovnici Rovnice 1. Determinant matice nesmí být roven 0.

Rovnice 1. Hillova šifra – ukázka převodu klíčové matice písmen na čísla

$$\begin{pmatrix} T & E & C \\ H & D & O \\ K & U & M \end{pmatrix} => \begin{pmatrix} 18 & 4 & 2 \\ 7 & 3 & 14 \\ 10 & 20 & 12 \end{pmatrix}$$

Otevřený text určený k šifrování je rozdělen do bloků o stejném počtu jako množství sloupců klíčové matice. OT se rovněž převede na čísla a vytvoří se vektor (viz Rovnice 2). V případě slova AHOJKY je rozděleno na AHO a JKY.

Rovnice 2. Hillova šifra – ukázka převodu OT do vektoru

$$\begin{pmatrix} A \\ H \\ O \end{pmatrix} => \begin{pmatrix} 0 \\ 7 \\ 14 \end{pmatrix} \qquad \begin{pmatrix} J \\ K \\ Y \end{pmatrix} => \begin{pmatrix} 9 \\ 10 \\ 24 \end{pmatrix}$$

Dále se klíčová matice vynásobí se získanými vektory a u každého se provede operace modulo 26. Výsledkem jsou zašifrované vektory, které se převedou zpět na písmena a vytvoří tak zašifrovanou zprávu pomocí Hillovi šifry (viz Rovnice 3).

Rovnice 3. Ukázka šifrování Hillovou šifrou

$$\begin{pmatrix} 18 & 4 & 2 \\ 7 & 3 & 14 \\ 10 & 20 & 12 \end{pmatrix} * \begin{pmatrix} 0 \\ 7 \\ 14 \end{pmatrix} = \begin{pmatrix} 56 \\ 217 \\ 308 \end{pmatrix} mod 26 = \begin{pmatrix} 4 \\ 9 \\ 22 \end{pmatrix} = \begin{pmatrix} E \\ J \\ W \end{pmatrix}$$

$$\begin{pmatrix} 18 & 4 & 2 \\ 7 & 3 & 14 \\ 10 & 20 & 12 \end{pmatrix} * \begin{pmatrix} 9 \\ 10 \\ 24 \end{pmatrix} = \begin{pmatrix} 250 \\ 429 \\ 578 \end{pmatrix} mod 26 = \begin{pmatrix} 16 \\ 13 \\ 6 \end{pmatrix} = \begin{pmatrix} Q \\ N \\ G \end{pmatrix}$$

Z původního textu tedy vzniká šifrový text EJW QNG.

Dešifrování probíhá vynásobením vektoru ŠT inverzní maticí k původní klíčové matici a provedením operace modulo 26. Tímto se získá zpět původní text. [17]

3.1.10 Polybiův čtverec a další možnosti substitučních tabulek

Polybiův čtverec je jednoduchá substituční šifra, ze které vychází spousta dalších, i některé zmíněné výše. Jedná se o klíčovou tabulku abecedy 5 x 5 polí a otevřený text je pouze nahrazován číselnými souřadnicemi řádku a sloupce.

Mezi další substituční tabulky patří například tabulka 5 x 10 (Tabulka 9), kde se kromě základní abecedy mohou nacházet i jiné specifické znaky, interpunkční znaménka apod. Při šifrování zde opět dochází pouze k nahrazení původního textu čísly souřadnic řádku a sloupce, jak lze vidět v tabulce Tabulka 10.

Tabulka 9. Klíčová tabulka 5 x 10 [18]

	1	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**0**
1	A	Á	B	C	Č	D	Ď	E	Ě	É
2	F	G	H	I	Í	J	K	L	M	N
3	Ň	O	Ó	P	Q	R	Ř	S	Š	T
4	Ť	U	Ů	Ú	V	W	X	Y	Ý	Z
5	Ž	;	„	-	/	,	+	?	!	

Tabulka 10. Ukázka šifrování pomocí klíčové tabulky 5 x 10

K	R	Y	P	T	O	L	O	G	I	E		Y	E	S	!
27	36	48	34	30	32	28	32	22	24	18	50	48	18	38	59

Existuje i šifrování nahrazováním otevřeného textu souřadnicemi v písmenkové podobě, a to s klíčovou tabulkou 3 x 13. V tabulce jsou kromě písmen umístěny i čísla a jiné znaky. Příklad je uveden v tabulce Tabulka 11. Text AHOJ/ZDRAVI99 lze zašifrovat jako ADAKBEAMCPBPAGBHADBLALCLCL. Pro přehlednost je možnost rozdělit text například po pěticích.

Tabulka 11. Klíčová tabulka 3 x 13 s písmenkovými souřadnicemi [18]

	D	**E**	**F**	**G**	**H**	**I**	**J**	**K**	**L**	**M**	**N**	**O**	**P**
A	A	B	C	D	E	F	G	H	I	J	K	L	M
B	N	O	P	Q	R	S	T	U	V	W	X	Y	Z
C	1	2	3	4	5	6	7	8	9	0	-	+	/

Původní zpráva může být také nahrazována i více možnostmi, kterými disponuje tabulka s vícenásobnými souřadnicemi (viz Tabulka 12). Použitím této tabulky lze dostat z OT AHOJ+ posílaný šifrový text ANGHKOJWFK, ale je možnost jej zašifrovat i například jako JADUEBDJCX. U každého znaku se nabízí 8 možností, jak jej lze šifrovat.

Tabulka 12. Klíčová tabulka s vícenásobnými souřadnicemi [18]

				A	**B**	**C**	**D**	**E**	**F**	**G**	**H**	**I**	**J**	**K**	**L**	**M**
				N	**O**	**P**	**Q**	**R**	**S**	**T**	**U**	**V**	**W**	**X**	**Y**	**Z**
A	**D**	**G**	**J**	A	B	C	D	E	F	G	H	I	J	K	L	M
B	**E**	**H**	**K**	N	O	P	Q	R	S	T	U	V	W	X	Y	Z
C	**F**	**I**	**L**	1	2	3	4	5	6	7	8	9	0	+	-	/

3.1.11 Autokláv

Šifra autokláv, jež byla navržena p. De Viaris, je pozměněná verze šifry pana Vigenéra. Dochází k využívání Vigenérova čtverce, ale klíč slouží pouze pro začátek šifrování původního textu. Neopakuje se neustále za sebou jako tomu je u Vigenérovy šifry. Jakmile je při šifrování vyčerpáno klíčové slovo, nastávají dvě možnosti, jak šifrování pokračuje. První z nich je využití jako dalšího klíče samotného šifrového textu. Druhou možností je jako pokračující klíč použít OT. Ukázky dvou možností šifrování jsou uvedeny v tabulkách Tabulka 13 a Tabulka 14. [18]

Tabulka 13. Autokláv – použití ŠT jako pokračujícího klíče [18]

Klíč	P	R	A	H	A	**T**	**S**	**G**
OT	D	N	E	S	B	Y	L	Z
ŠT	**T**	**S**	**G**	I	C	R	O	J

Tabulka 14. Autokláv – použití OT jako pokračujícího klíče [18]

Klíč	P	R	A	H	A	**D**	**N**	**E**
OT	**D**	**N**	**E**	S	B	Y	L	Z
ŠT	T	S	G	I	C	N	X	W

3.2 Transpoziční šifry

Jedná se o šifry klasické kryptografie. Oproti substituci nenahrazují původní abecedu abecedou šifrovanou, ale dochází ke změně pořadí znaků v textu. Hodnoty jednotek se ponechávají beze změny. Prvotní užívání transpozičních šifer sahá pravděpodobně do starověkého Řecka kolem roku 400 před naším letopočtem, kdy byl pro přenos zpráv užíván text na pruhu kůže, který dával smysl po ovinutí na tyč o určitém průměru.

3.2.1 Skytale

Jedná se o historicky první šifrovací metodu využívanou ve válkách. Sparťané využívali tyče o různém průměru, kolem kterých obmotali materiál, na který bylo možné zapisovat. Zpráva byla na obmotaný proužek napsána klasicky po řádcích a poslána určené osobě. Ta mohla zprávu přečíst jen s vědomostí průměru dané tyče, kterou odesílatel použil. [3]

3.2.2 Šifra se zápisem mezi řádky

Jedná se o jednoduchou transpoziční šifru, při které dochází k zapisování původního textu střídavě mezi řádky. Lze říci, že se písmena zapisují postupně do sloupců. Šifrový text je ten, který je poté čten postupně po řádcích.

Ukázka šifrování textu VASE OBJEDNANE ZBOZI DODAME ZÍTRA NA SMLUVENE MÍSTO pomocí tří řádků je uvedena v tabulce Tabulka 15. Šifrovaný text je nyní přečten jako VEJNE ODAZR ALEMT AOEAZ ZOMIA SUNIO SBDNB IDETN MVES.

Tabulka 15. Ukázka transpoziční šifry se zápisem mezi 3 řádky

V	E	J	N	E	O	D	A	Z	R	A	L	E	M	T
A	O	E	A	Z	Z	O	M	I	A	S	U	N	I	O
S	B	D	N	B	I	D	E	T	N	M	V	E	S	

3.2.3 Zápis OT do tabulky s klíčovým slovem

Tato šifra dříve patřila k velmi oblíbenému způsobu utajování informací. Je zde využito klíčového slova a zápisu OT po řádcích do tabulky. Jakmile je text zapsán, sloupce se seřadí podle posloupnosti písmen klíčového slova v abecedě. Výsledkem je zašifrovaný text. Ukázka zápisu do tabulky s následným přehozením sloupců je zobrazena v tabulkách Tabulka 16 a Tabulka 17 .

V první tabulce je možné vidět klíčové slovo KLIC na prvním řádku. Vedle písmen jsou v závorkách čísla, která odpovídají posloupnosti daných písmen v abecedě. V řádcích pod klíčem je zapsán otevřený text – AHOJ JAK SE MAS. Druhá tabulka již znázorňuje přesunutí celých sloupců podle čísel u znaků klíče. Šifrový text je tentokrát čten po sloupcích – JSSOKA AJEHAM.

Tabulka 16. Transpoziční šifra – tabulka s klíčovým slovem

K (3)	L (4)	I (2)	C (1)
A	H	O	J
J	A	K	S
E	M	A	S

Tabulka 17. Transpoziční šifra – tabulka s klíčovým slovem po zašifrování

C (1)	I (2)	K (3)	L (4)
J	O	A	H
S	K	J	A
S	A	E	M

Jestliže je potřebné ještě více zkomplikovat práci kryptoanalytikům, je možné zašifrovat již šifrový text ještě jednou pomocí dalšího klíče, zapsáním do tabulky a přehozením sloupců. Příjemce takto zašifrované zprávy bude znát velikost obou tabulek a dvě klíčová slova. Dešifrování probíhá pouze opačným způsobem. Do tabulky je zapsán zašifrovaný text po sloupcích a příjemce sloupce seřadí podle písmen v klíči, nikoli podle posloupnosti daných znaků v abecedě. [19]

3.2.4 Šifrování mřížkou

Autor šifrování textu za pomocí mřížky, Jeroným Cardan, pocházel z dob renesance 16. století. Mřížka byla ve své době často aplikovanou šifrou. Mnozí diplomaté a plukovníci ji postupem času neustále zdokonalovali. V Cardanově mřížce jsou obsažena i písmena, která nejsou součástí OT, tzv. klamače. Často jsou zapsány jiným typem písma než právě OT. Mřížka obsahuje 63 polí (jde tedy o tabulku s rozměry 9 x 7), z nichž 8 znaků je právě klamačů. OT se zapisuje do mřížky po řádcích. Ty pole, která obsahují klamače, jsou přeskočena. ŠT je získán přečtením/zapsáním textu z mřížky po sloupcích. Ukázku použití mřížky lze vidět v tabulce Tabulka 18. V tomto případě je OT NIC NENÍ TAK TEZKE ABY SE TOHO DAREBACTVIM NEDOSAHLO XPERTONIUSX zašifrován jako NABDV YTIKY AIARC OSRAH atd.

Tabulka 18. Transpozice – mřížka [18]

N	I	C	N	u	E	N	I	T
A	K	o	T	E	Z	K	E	A
B	Y	S	E	e	T	O	H	O
D	A	R	E	B	A	i	C	T
V	I	a	M	N	E	D	O	S
y	A	H	L	O	X	P	E	e
T	R	O	N	I	a	U	S	X

3.2.5 Zubatka

Zubatka je používána zejména pro texty, při jejichž zápisu do tabulky je délka sloupců v tabulce větší jak použité klíčové slovo. Tabulka je rozdělena podle pořadí znaků klíčového slova v abecedě. Vznikají 2 části tabulky. OT je zapsán po řádcích, a to nejprve do první části tabulky a poté až do části druhé. Pole, která nejsou zaplněna OT, jsou doplněna náhodnými písmeny z abecedy. Šifrový text se dostává přečtením tabulky po sloupcích. Příklad šifrování Zubatkou je uveden v tabulce Tabulka 19.

Tabulka 19. Transpozice – Zubatka [18]

	S (10)	**E (4)**	**D (3)**	**M (8)**	**I (5)**	**K (6)**	**R (9)**	**A (1)**	**S (11)**	**K (7)**	**A (2)**
1	V	P	R	A	T	E	L	S	T	V	I
2	N	E	L	Z	E	N	I	R	C	P	O
3	V	A	Z	O	V	A	T	I	Z	A	T
4	Z	H	A	O	U	B	N	K	E	J	A
5	S	V	A	I	N	E	Z	N	P	O	I
6	C	X	C	H	I	L	E	C	B	O	E
7	V	R	O	A	X	P	N	O	I	L	T
8	I	A	J	C	I	P	H	R	O	I	T
9	C	E	L	E	N	A	E	P	N	O	M
10	I	I	N	E	J	A	L	E	A	V	E
11	R	E	J	N	E	C	H	V	P	A	L
12	X	S	E	N	E	C	A	X	X	X	X

V prvním řádku je vypsáno klíčové slovo SEDMIKRÁSKA s číslem, které označuje posloupnost písmen podle abecedy. Pokud jsou v klíči obsažena 2 stejná písmena, jsou označena dvěma různými čísly. Každý sloupec tabulky je rozdělen na 2 části podle hodnoty těchto čísel, což je v tabulce znázorněno i graficky. [18]

3.3 Sloučení substituční a transpoziční šifry

3.3.1 ADFGX, ADFGVX

Je založena na modifikovaném substitučním čtverci s kombinací sloupcové transpozice. Tyto dvě šifry jsou téměř totožné. Hlavním rozdílem je velikost tabulky a u šifry ADFGVX použití jednoho druhu písmene (V) navíc. ADFGX byla používána za první světové války německou armádou. Jednalo se o tzv. polní šifru. Její název je odvozen od písmen, která jsou použita v šifrovém textu. Právě tato písmena byla vybrána z důvodu různě znějícího a dobře slyšitelného zvuku při přenosu prostřednictvím Morseova kódu. Účelem bylo zabránit chybám přenosu a osoby, která jej zajišťovala.

Při šifrování ADFGX se nejprve zvolí klíčová tabulka 5 x 5 obsahující náhodně uspořádaná všechna písmena abecedy kromě písmene J, které je nahrazeno znakem I. Dále je voleno

klíčové slovo potřebné pro transpozici. OT se převede na písmena souřadnic z klíčové tabulky v pořadí řádek, sloupec. Každé písmeno je tudíž nahrazeno dvěma znaky. Dále dochází k transpozici. Pod zvolený klíč se do řádků vypíšou získané souřadnice OT. Probíhá zde sloupcová transpozice, kdy jsou písmena klíčového slova seřazena podle posloupnosti v abecedě a přesouvají sebou i sloupce textu. Výsledkem je šifrový text, který je z tabulky čten po sloupcích. Ukázka šifrování slova AHOI za použití šifry ADFGX je znázorněna v tabulkách níže. [20]

Tabulka 20. Šifrovací tabulka ADFGX

	A	**D**	**F**	**G**	**X**
A	a	f	n	c	i
D	o	b	h	r	l
F	g	m	s	d	q
G	p	t	v	x	u
X	e	y	z	w	k

Po převedení slova AHOI podle zvolené klíčové tabulky (Tabulka 20) vychází souřadnice textu AA DF DA AX. Tyto souřadnice zapsány do tabulky s klíčovým slovem lze vidět v tabulce Tabulka 21.

Tabulka 21. ADFGX – tabulka s klíčovým slovem

K	**L**	**I**	**C**
A	A	D	F
D	A	A	X

Konečné seřazení sloupců tabulky podle posloupnosti písmen klíčového slova zobrazuje Tabulka 22. Šifrovaný text čten po sloupcích je ve tvaru FX DA AD AA.

Tabulka 22. ADFGX – tabulka s klíčovým slovem seřazena dle posloupnosti písmen klíče

C	**I**	**K**	**L**
F	D	A	A
X	A	D	A

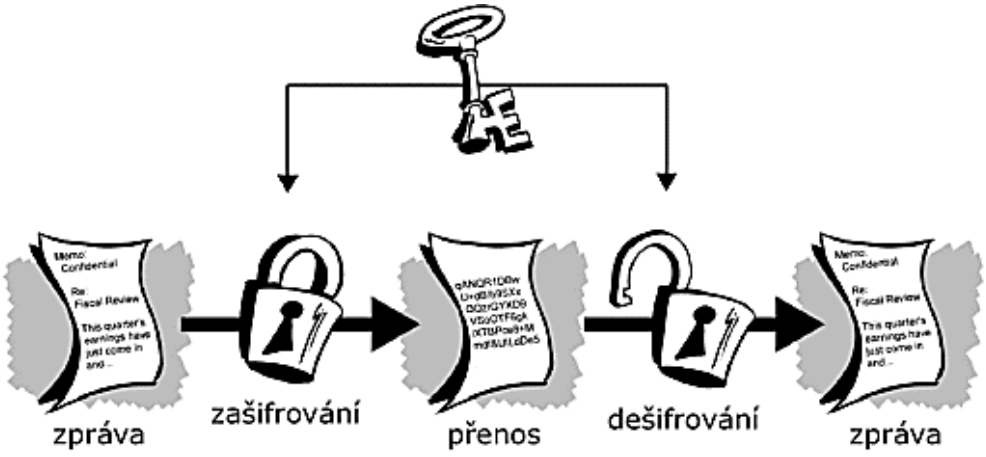
Šifra ADFGVX se zakládá na tabulce o velikosti 6 x 6 a zahrnuje náhodně uspořádanou abecedu 26 znaků a čísla od 0 po 9. Řádek i sloupec mají oproti šifře ADFGX jeden identifikátor souřadnic navíc, a to písmeno V. Jinak šifrování probíhá stejně jako u šifry předchozí.

Aby mohli být tyto šifry bez větších problémů dešifrovány, příjemce musí znát detailní informace o uspořádání znaků v tabulkách a klíčové slovo užívané při transpozici. [3]

4 MODERNÍ KRYPTOGRAFIE

Moderní kryptografie slouží zejména k pokročilejšímu zabezpečování počítačů a komunikace obecně. Nenahrazuje přímo písmena textu, ale většinou pracuje s jejich vyjádřením pomocí binárních čísel. Využívá složitější matematické operace, jako je například teorie čísel, teorie výpočetní složitosti apod. I v oblasti moderního šifrování jsou zastoupeny druhy šifrovacích algoritmů s rozdílnou složitostí a zabezpečením. Rozlišují se dvě základní skupiny, a to kryptografie symetrických a asymetrických klíčů. Rozdílem je počet klíčů a způsob jejich využívání. Dále dochází k využívání hash funkcí nebo hybridního šifrování.

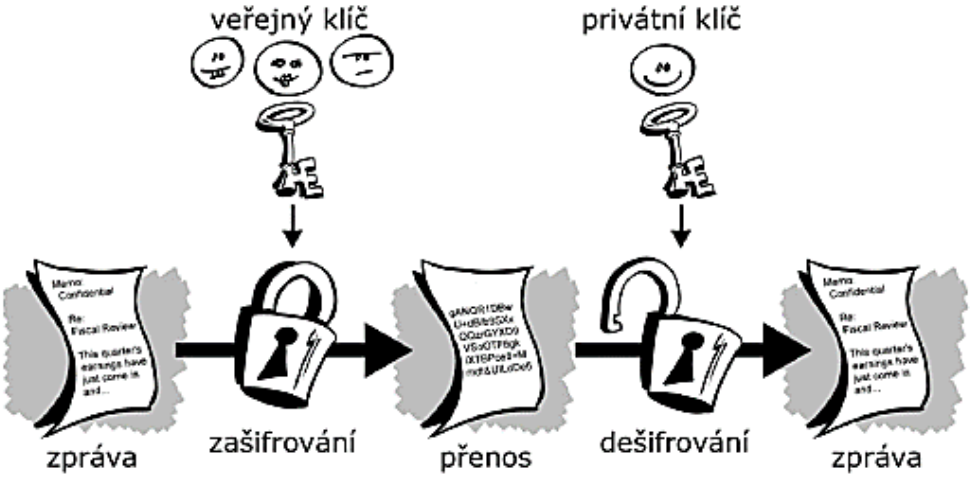
Výhodou symetrických systémů je, že jsou rychlejší a jednodušší. Jsou méně náročné na výpočetní výkon než šifrování asymetrické. Problém nastává u výměny klíče mezi odesílatelem a příjemcem. Je k dispozici pouze jeden klíč, který slouží pro šifrování i dešifrování zprávy, tudíž musí docházet k jeho sdílení, což není nejbezpečnější. Princip šifrování je znázorněn na obrázku Obrázek 7. Bezpečnost zde závisí zejména na délce klíče. Čím delší klíč je zvolen, tím se jeho náhodné uhodnutí stává náročnějším. Klíče s délkou 256 bitů jsou považovány za relativně bezpečné a odolné. V symetrickém šifrování se nejčastěji používají blokové a proudové šifry. Blokové šifry shromažďují data v blocích (skupinkách) s určenou velikostí a každý blok je následně šifrován pomocí klíče a algoritmu. Proudové šifry data nešifrují po blocích, ale dochází k šifrování po jednotlivých bitech. Mezi symetrické šifry jsou řazeny například RC4, AES, DES, 3DES a Blowfish.



Obrázek 7. Princip symetrického šifrování [21]

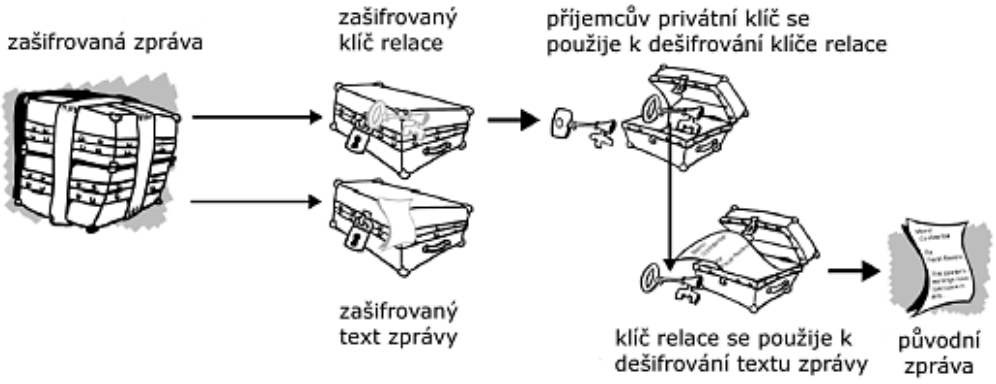
Asymetrické systémy jsou založeny na využití dvojice klíčů (veřejný a soukromý). Veřejná část klíče je většinou dostupná všem a slouží k zašifrování zprávy. Soukromou část klíče vlastní pouze příjemce, který si je tak schopen zprávu dešifrovat. Obvykle je použit stejný

algoritmus jak pro šifrování, tak pro dešifrování a mění se pouze klíč, který je aplikován. Tato metoda je novější a bezpečnější. Výhodou je, že nemusí docházet ke sdílení jednoho klíče jak u šifer symetrických. Ovšem i asymetrické šifry mají negativum – jelikož se zde aplikují složité matematické operace, jsou poněkud pomalejší a méně praktické pro větší datové přenosy než šifry symetrické. Mezi asymetrické šifry jsou řazeny například RSA a Diffie – Helman. Princip asymetrického šifrování na základě dvojice klíčů je uveden na obrázku Obrázek 8.



Obrázek 8. Princip asymetrického šifrování [21]

Z důvodu nevýhod asymetrické techniky zmíněných výše dochází ke kombinování symetrického a asymetrického šifrování zvaného hybridní šifrování. Zde se využívá rychlost symetrického systému spolu s použitelností a bezpečností systému asymetrického. Zpráva je zašifrována symetrickým klíčem, který je poté zašifrován klíčem veřejným. Takto zašifrovaný klíč je poslán spolu se zašifrovanou zprávou příjemci. Aby příjemce dostal zpět původně použitý symetrický klíč, musí jej dešifrovat svým soukromým klíčem. Následně klíč může být použit k dešifrování zprávy. [21][22]



Obrázek 9. Princip hybridního šifrování [21]

4.1 DES

Na počátku 80. let minulého století došlo ke zlomu, kdy se kryptografie začala používat i pro komerční účely, například v bankovnictví. Do této doby bylo šifrování většinou drženo v tajnosti vlády. Nejprve byl vyvinut algoritmus, který byl založen na šifře Lucifer ze 70. let. Lucifer patřil mezi první blokové šifry, které pracovaly s digitálními daty.

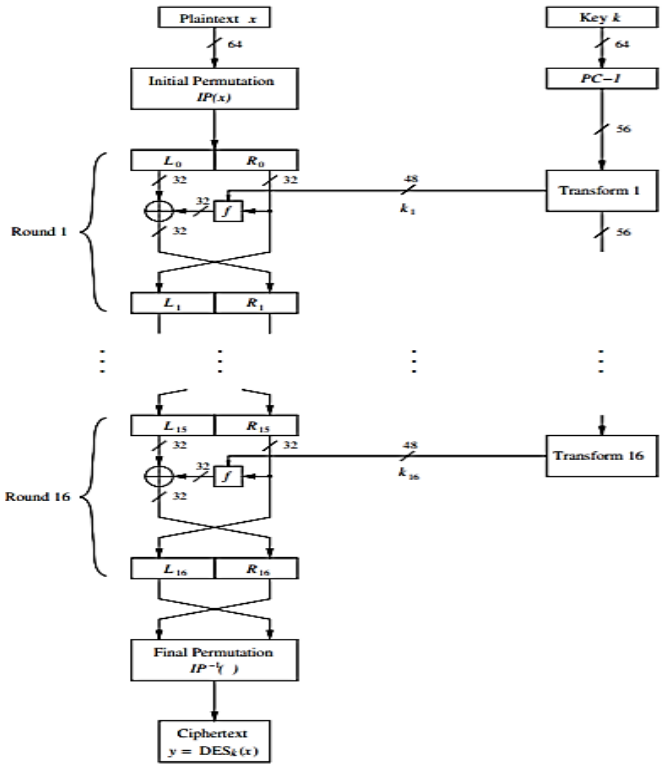
DES (Data Encryption Standard) se stal nejoblíbenější symetrickou blokovou šifrou přibližně od 90. let. Dnes již ale klasická verze DES není příliš bezpečná, protože nemá dostatečně velký klíčový prostor. Bezpečnější je verze, kdy dochází k šifrování dat třikrát za sebou – 3DES. Šifra DES je velmi dobře prostudovaná šifra, která se stala inspirací pro mnohé současné šifry. Na konci minulého století začala být postupně nahrazována AES.

Aby bylo dosaženo silného šifrování, používá DES 2 základní operace:

- „Confusion“ – snaží se skrýt vztah mezi klíčem a šifrovým textem. Běžným nástrojem pro dosažení této operace je využití substituce.
- „Diffusion“ – změna jednoho znaku původního textu vyvolá změnu více znaků na výstupu s cílem skrýt statistické vlastnosti původního textu. Často se jako difuzní prvek využívá bitová permutace.

Šifra DES je složena z kol operací, které se na data opakovaně aplikují. Šifruje bloky o velikosti 64 bitů s pomocí klíče o velikosti 56 bitů. Každý blok původního textu je zpracován v 16 kolech, kde probíhají stejné operace. Pro každé kolo je použit jiný klíč, který je vždy odvozen od klíče hlavního. DES užívá Feistelovu síť, což umožňuje téměř totožné šifrování i dešifrování. Při dešifrování je pouze vyžadováno obrácené plánování klíče. Po vstupu bloku do šifry je provedena počáteční bitová permutace – text je rozdělen na 2 poloviny. Obě poloviny prochází dalšími 16 kroky. Do kroků s nimi vstupuje i klíč, který je původně složen ze 64 bitů, tudíž musí být upraven na bitů 56, a to odstraněním každého 8. bitu. V každém kroku je do funkce *f* (zde dochází k operacím confusion a diffusion) přiváděna pravá polovina bloku. Výstup funkce *f* je spojen s levou polovinou bloku (značeno $\oplus$). Nakonec dochází k záměně levé a pravé poloviny bloku a pokračuje se dalším kolem, kde se všechny operace opakují. Z 56bitového klíče se během každého kola generuje 48bitový dílčí klíč pomocí transformace klíče. Klíč je rozdělen na dvě poloviny. Máme dva 28bitové klíče. Tyto dvě poloviny jsou posunuty doleva o jednu nebo dvě polohy. To záleží, do jakého kroku se vstupuje. Po posunu dochází k výběru 48 z 56 ti bitů. Při transformaci klíče dochází k permutaci a výběru 48bitového klíče. Díky takovému výběru klíčů, se v každém kroku DES

používá jiná skupinka klíčových bitů. Feistelova struktura šifruje pouze levou polovinu bloku, pravá je vždy pouze zkopírována do dalšího kola. Nakonec dochází k opětovnému spojení polovin bloku a konečné permutaci. Výsledkem je opět 64bitový šifrový text. Feistelova struktura šifry DES je zobrazena na obrázku Obrázek 10. [23]



Obrázek 10. DES – Feistelova struktura [23]

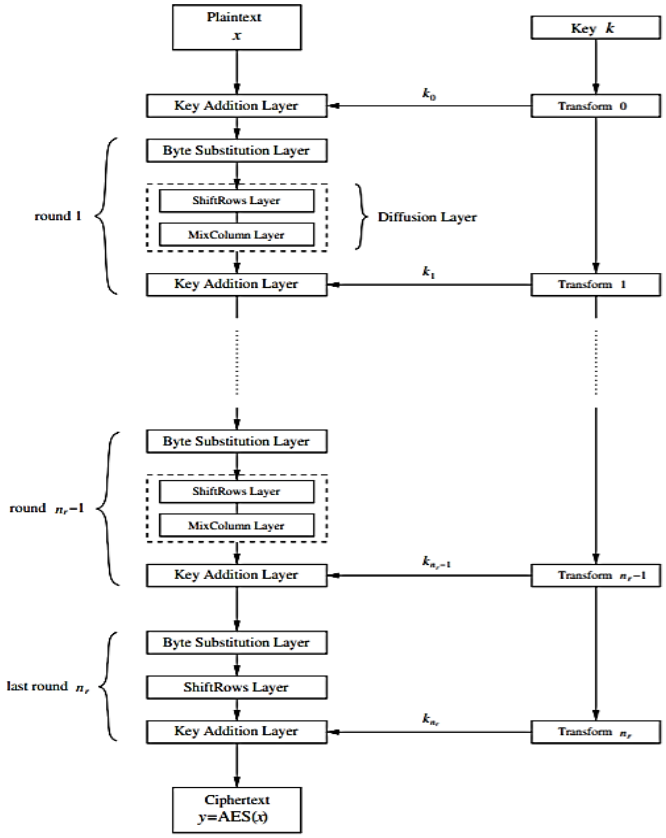
4.2 AES

AES je rychlá a bezpečná forma šifrování, která postupem času nahradila DES a stala se celosvětovým standardem. Koncem 20. století se začala zpochybňovat bezpečnost šifry DES zmíněné výše a americká vláda nalezla nový standart, a to právě AES. Roku 2002 byl schválen v USA a rychle se dostal do celého světa. Pro zajímavost se používá i například k šifrování Wi-Fi nebo zabezpečení Skype.

AES patří mezi blokové šifry. Šifruje bloky o velikosti 128 bitů pomocí klíče o 3 podporovaných velikostech (128, 192 nebo 256 bitů). Proces šifrování je prováděn v binárních číslech a operace jsou matematicky složitější. AES je složena ze tří vrstev a každá vrstva pracuje s celým 128bitovým blokem dat. Podobně jako šifra DES i zde data prochází při šifrování několika koly. Každé kolo kromě prvního a posledního obsahuje všechny 3 vrstvy. V posledním kole se neprovádí transformace MixColumn. Tři vrstvy, přes které data jdou:

- „Key Additional“
- „Byte Substitution“
- „Diffusion – ShiftRows a MixColumn“

Nejprve dochází k rozšíření klíče. Jsou odvozeny i další dílčí klíče pro užití v průběhu šifrování. V prvním kole je šifrování prováděno pomocí počátečního klíče, který je přidán do bloku zprávy za pomoci operace XOR (vrstva Key Addition). V ByteSub vrstvě dochází k nelineární transformaci každého prvku (bitů) pomocí předdefinovaných tabulek s matematickými vlastnostmi. Zajišťuje šíření změn. Představuje operaci Confusion zmíněnou výše u DES šifry. Difuzní vrstva je složena ze dvou podvrstev, které provádí lineární operace. Vrstva ShiftRows permutuje data na úrovni bytů. Dochází k postupnému posouvání řádků doprava. Druhý řádek se posouvá o 3 byty vpravo, třetí o dva atd. Ve vrstvě MixColumn míchá každý sloupec stavové matice. Tyto operace se několikrát opakují (viz Obrázek 11). Počet opakování závisí na velikosti klíče. U 128bitového klíče jsou operace prováděny 9x, u klíče s velikostí 256 bitů jsou vrstvy opakovány až po 13 kol. U posledního kola je vynechána operace MixColumn a výsledkem je šifrový text. [23]



Obrázek 11. AES – diagram průběhu šifrovaní [23]

4.3 RSA

RSA byla navržena roku 1977 poté, co Diffie a Hellman představili světu kryptografii veřejného klíče. Na vytvoření šifry spolupracovali Ronald Rivest, Adi Shamir a Leonard Adleman. Šifrování pomocí RSA se stalo jednou z nejpoužívanějších asymetrických šifer. Využívá se například při digitálním podpisu nebo pro digitální certifikáty. Cílem šifry není nahrazení symetrického šifrovaní, protože je RSA pomalejší a výkonově náročnější. V praxi je RSA často používáno se šifrou symetrickou, kdy jsou data šifrována klíčem symetrickým, který je následně zašifrován asymetrickým způsobem.

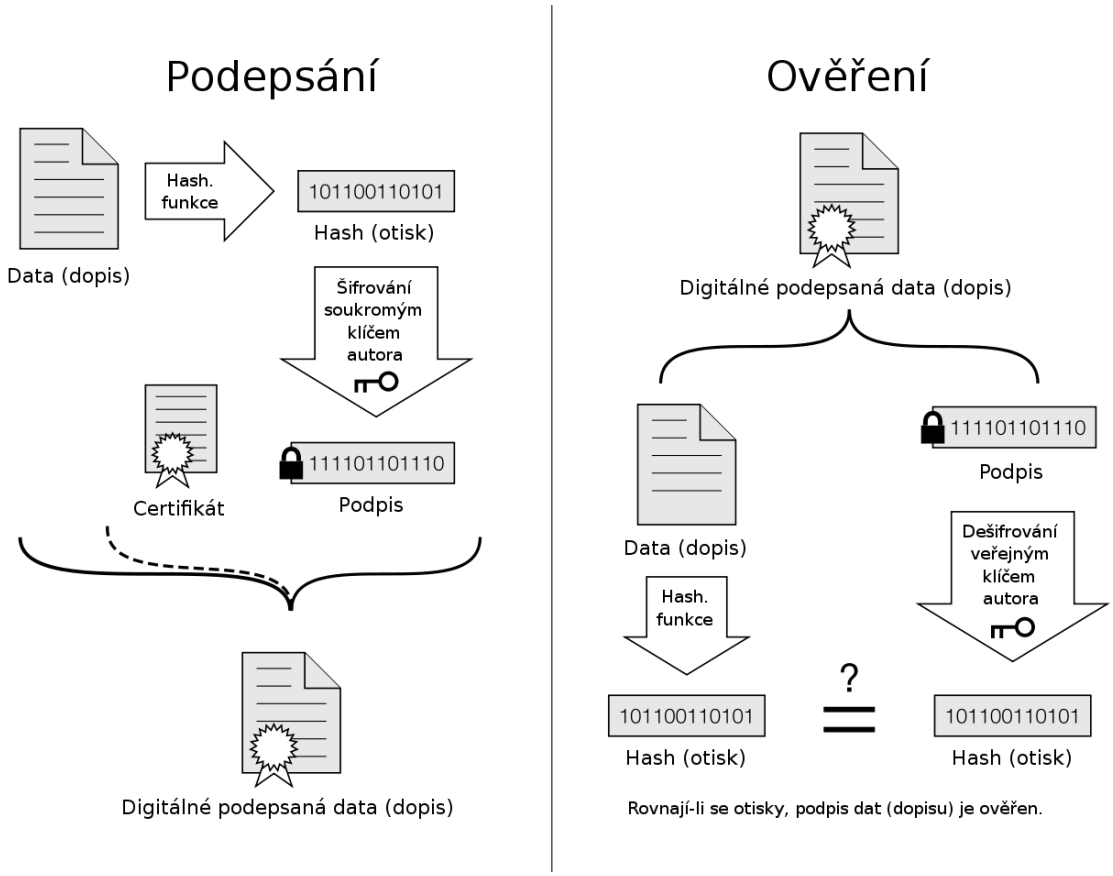
RSA šifra pracuje s veřejným a soukromým klíčem. Jelikož se jedná o asymetrický systém, problém sdílení klíče před zahájením komunikace je vyřešen. Data jsou šifrována pomocí veřejné části klíče, která je přístupná veřejnosti. Jakmile je zpráva zašifrována, příjemce ji může dešifrovat pouze svým soukromým klíčem. Příjemce si vygeneruje dvojici klíčů. Veřejnou část sdílí s ostatními, soukromou si ponechá skrytou.

V průběhu šifry se používají vysoká prvočísla. Například v 2048bitové RSA šifře by výsledek prvočísel vytvořil klíče o velikosti 617 číslic. Algoritmus RSA funguje za předpokladu, že lze provádět výpočty pouze v jednom směru. Ze dvou prvočísel lze dostat výsledek, ale z výsledku je již velmi obtížné, ne-li nemožné, získat původní prvočísla nazpět. V prvním kroku šifrování se generují dvě náhodná velká prvočísla (p a q), díky kterým je později získán klíč. V dalším kroku dochází k výpočtu modulu n pomocí vzorce: $n = p \times q$. Dále probíhá výpočet $\lambda (n) = lcm (p-1, q-1)$, jehož výsledkem je Carmichaelův součet pro modul n. *Lcm* představuje nejmenší možný společný násobek. Po získání Carmichaelova součtu daných prvočísel se generuje veřejná část klíče (čísla *e* a *n*) – je potřeba určit hodnotu čísla *e*, která musí být mezi číslem 1 a hodnotou $\lambda (n)-1$. $Gcd(e, \lambda (n))$ musí být rovno 1, což značí že tato dvě čísla jsou nesoudělná a existuje inverze čísla *e* potřebná k získání soukromé části klíče. Šifrový text je výsledkem vzorce OT^e *modulo n.* Pro zpětné dešifrování textu je potřeba vygenerování soukromého klíče (číslo *d* a *n*). Hodnota čísla *n* je známá z předchozího postupu. Je tedy nutné určit číslo *d*, a to ze vzorce $d = e^{-1}$ *modulo* $\lambda (n)$. Původní text je získán za pomocí výpočtu $ŠT^d$ *modulo n.* Jakmile má odesílatel k dispozici veřejnou část klíče, kterou nasdílel příjemce, je schopen zašifrovat data a poslat je příjemci. Jedině ten si může data dešifrovat zpět, jelikož vlastní i soukromý klíč. [23]

4.4 Digitální podpis

Digitální podpis patří mezi důležité kryptografické nástroje používané i v dnešní době. Společné s ručně psaným podpisem má to, že poskytuje metodu zajišťující, že zpráva byla poslána právě jedním daným uživatelem tvrdícím, že ji poslal. Její autor také nemůže zpochybnit, že dokument odeslal právě on. Aby dokument mohla podepsat pouze osoba, která jej vytvoří a odešle, je podpis realizován pomocí soukromého klíče a spolu s dokumentem zaslán příjemci. Přijímací strana používá pro ověření podpisu klíč veřejný. Je tomu naopak než například u RSA šifrování. Digitální podpis také umožňuje kontrolovat integritu dokumentu (zda je doručená zpráva ve stejném stavu jako při odeslání).

Vůbec s první myšlenkou digitálních podpisů přišli Diffie a Hellman v 80. letech. Se zavedením RSA šifry se mohlo pomalu začít s jeho vytvořením. V 90. letech se digitální podpisy začaly využívat v praxi. Obecně se algoritmus digitálního podpisu musí skládat z algoritmu generování klíčů, algoritmu pro podepisování dat a algoritmu pro ověření podpisu. Prvním algoritmem užívaným ke generování digitálního podpisu bylo RSA. Princip vytvoření digitálního podpisu je znázorněn na obrázku níže (Obrázek 12).



Obrázek 12. Digitální podpis [24]

Nejprve se pomocí algoritmu (například RSA) vygeneruje dvojice klíčů. Dokument, který je určen k podpisu, se nechá projít hash funkcí. Tato funkce dokument nijak nešifruje, pouze vrací jeho číselnou hodnotu. Pro daný vstup vrací vždy jeden totožný výstup. Dva odlišné dokumenty nemohou mít stejný výstup hash funkce a z výstupu nelze dostat původní vstup. Jakmile je k dispozici výstup z hash funkce dochází k jeho šifrování pomocí soukromého klíče RSA šifrou. Takto zašifrovaný hash je společně s dokumentem poslán příjemci. Pokud chce příjemce ověřit autenticitu dokumentu nebo to, že dokument nebyl změněn, musí dešifrovat přijaté hash znaky pomocí veřejného klíče RSA šifrou. Dále dojde ke spočítání hashe z doručeného dokumentu. Pokud se tyto dvě hodnoty hash funkce rovnají, s dokumentem nebylo manipulováno a je všechno v pořádku. Pokud soukromý klíč nebyl ohrozen, jsou digitální podpisy považovány za bezpečné. [23] [25]

5 MOŽNOSTI OPTIMALIZACE KÓDU A TESTOVÁNÍ VÝKONU

Optimalizace nám poslouží zejména pokud je požadováno, aby aplikace či nějaký program běžel rychleji. U jednoduchých programů si uživatel mírného zlepšení rychlosti nemusí ani všimnout, může se jednat například pouze o milisekundy zlepšení. Pokud se však jedná o aplikaci, kde neustále narůstají požadavky a rozrůstá se stále i kód, rychlost může klesat více než je vhodné. Vhodná optimalizace kódu je schopna ušetřit i několik sekund (například při načítání stránky nebo zasílání požadovaných dat). Při psaní kódu se již samozřejmě mohou dodržovat alespoň některé principy, které nám rychlost zvýší. Abychom po spuštění programu věděli, kde se program zdržuje nejvíce, je vhodné také otestovat jeho výkon, a poté jej případně optimalizovat podle potřeby. V následujících podkapitolách je uvedeno pár tipů, jak dosáhnou optimalizovanějšího kódu v jazycích Python, Java a C#.

Dále je vhodné dodržovat obecné zásady pro psaní přehledného tzv. čistého kódu. To je oceněno především při spolupráci na projektech, kdy je programátorovi přidělen již rozpracovaný projekt a měl by se v něm vyznat nebo při nabalování stále dalších funkcí, které vyplynou z požadavků zákazníka. Nemělo by docházet, že při přidání další funkcionality se rozpadne jiná část kódu. Program by měl být co nejjednodušeji upravitelný, přehledný, elegantní. Mělo by být jasné, co každý kousek kódu provádí. Samozřejmě i „úklid“ kódu si žádá určité časové prostředky, které je nutné vynaložit, ale většinou se to zejména z dlouhodobého hlediska vyplatí. Kód by se měl vyhýbat duplicitním částem. Každá proměnná, třída nebo funkce by měla být pojmenována tak, aby bylo jasné k čemu slouží nebo co obsahuje. Když například bude vytvořena proměnná *int i*, nikdo bez většího bádání kódu nebude hned vědět, co tato proměnná bude uchovávat. Pokud je místo toho vytvořena proměnná *int mnozstviKusuProduktu*, je to poněkud jasnější. Při psaní funkcí by se mělo dbát na to, aby byly co nejkratší, řešily pouze jednu věc a obsahovaly co nejmenší počet argumentů. Logickým argumentům (true/false) bychom se měli vyhnout. Pokud funkce vyžaduje vyšší počet argumentů, je vhodné je předat pomocí třídy (jejího vytvořeného objektu). Dále je místo ověřování nějakého stavu pomocí *if* a následné vracení chyby lepší použít bloky *try/catch*, se kterými lze kód opět zjednodušit. Komentáře by se neměli snažit zlepšit přehlednost nepřehledného kódu. Vždy je lepší kód upravit, použít výstižná jména a komentáři pouze vypomoct dovysvětlit činnost kódu. Komentáře mohou sloužit také jako informace, co se má v kódu/funkci doimplementovat. Měly by být jasné i pro jiného programátora. Pokud je jisté, že řádky kódu nebudou potřebné, měly by být smazány, ne pouze zakomentovány. Později takový komentář může být matoucí.

Kvůli přehlednosti je také dobré kód vhodně formátovat. Například každá funkce může být oddělena od ostatních prázdným řádkem, a naopak řádky souvisejícího kódu hned pod sebou. Deklarační proměnné je vhodné deklarovat co nejblíže části, ve které jsou použity. Naopak instanční proměnné je dobrým zvykem deklarovat hned na začátku třídy. Funkce, která je volána z jiné funkce, by měla být v její blízkosti, nejlépe pod ní. Velmi užitečným nástrojem je taktéž použití odsazování řádků. Podřízené bloky by měly být oproti blokům nadřízeným odsazeny o jeden řád doprava. Při používání různých knihoven a balíčků je také potřeba zachovat obezřetnost. Může se stát, že se postupem času změní, a bude tak potřeba udělat úpravy kódu na mnoha místech. Proto je někdy výhodné si vytvořit novou třídu, která se bude například starat o převody typů a zajistí správné fungování pouze na jednom místě (Obrázek 13). Toto jsou pouze některé z vlastností pro psaní čistého kódu. [32]

```
public class Sensors {
  private Map sensors = new HashMap();
  public Sensor getById(String id) {
    return (Sensor) sensors.get(id);
  }
  //část kódu
}
```

Obrázek 13. Nová třída použití knihovny [32]

5.1 Tipy pro optimalizaci – Python

Jednou z možností optimalizace je List comprehension (viz Obrázek 14). Jedná se o syntaktickou konstrukci, kterou je vhodné využít v případě vytváření listu. Například když je definován jeden list a druhý má mít stejnou velikost jako první. V případě, že v prvním listu je kladné číslo, ve druhém bude 1, pokud ne, do druhého listu se na daný index vloží 0. Využití List comprehension je rychlejší než například využívaní smyčky for spolu s podmínkami ověřování, co se právě nachází na indexu v prvním listu.

```
#pomalejsi
prvni_list=[-2,2,9]
druhy_list=[]

for i in prvni_list:
        if i>0:
            druhy_list.append(1)
        else:
            druhy_list.append(0)


print (druhy_list[2])

#rychlejsi
prvni_listC=[-2,2,9]
druhy_listC=[1 if i > 0 else 0 for i in prvni_listC]
print (druhy_listC[2])
```

Obrázek 14. Python – List comprehensions

I volání funkcí je v Pythonu poněkud náročné. Je dobré promýšlet, na co jsou funkce opravdu potřeba. Tady ale nastává otázka, zda musí být upřednostněna rychlost před přehledností a testovatelností kódu. Je vhodné volit vestavěné funkce, pokud je to tedy možné, jako *sum()*, *max()*, *map()*, *str.join()*, *sorted()* a další. Vestavěné funkce se budou provádět rychleji, než kdybychom je napsali sami (Obrázek 15).

```
#pomalejsi
slova=['Toto','jsou','slova','na','pridani']
veta=''
for slovo in slova:
    veta+=slovo+' '
print(veta)
#rychlejsi
slova2=['Toto','jsou','slova','na','pridani']
veta2=' '.join(slova2)
print(veta2)
```

Obrázek 15. Python – použití vestavěné funkce join()

V Pythonu je také možnost využít zápisu *x,y = y,x*, kdy dojde k prohození proměnných. Je to elegantnější zápis a také nám ušetří nějaký ten čas, než použití třetí proměnné na uložení a prohození požadovaných dvou proměnných *x* a *y* (viz Obrázek 16).

```
#pomalejsi
x=10
y=5
a=x
x=y
y=a
print(x,y)
#rychlejsi
x=10
y=5
x,y=y,x
print(x,y)
```

Obrázek 16. Python – prohození proměnných

Jestliže je potřeba definovat například proměnnou, ve které je obsažena proměnná jiná, je možné provést zápis viz Obrázek 17.

```
#pomalejsi
castA="Markéta"
castB="Jmenuji se " + castA + "."
print(castB)
#rychlejsi
castA2="Markéta"
castB2="Jmenuji se %s." % castA2
print(castB2)
```

Obrázek 17. Python – zápis proměnné pomocí %s

Pokud je to možné je vhodné používat lokální proměnné místo globálních. Lokální proměnná je načtena rychleji. [26][27]

5.2 Tipy pro optimalizaci – Java

Jakmile je napsán program/zdrojový kód v jazyce Java dochází k jeho přeložení do bytového kódu, který již může sloužit jako instrukce pro JVM (Java Virtual Machine – provádí instrukce). JVM obsahuje ještě kompilátor JIT (Just-In-Time). Ten je schopen přeložit bytový kód do nativního kódu, který se skládá z instrukcí jasně definovaných pro typ procesoru, na kterém poběží. Pouhé využití JIT překladače je bráno jako optimalizování zpracovávání kódu, jelikož nativní kód se provádí rychleji. Samozřejmě je nutné zvážit, kde všude je potřeba převodu do nativního kódu, jelikož převod zabere taky nějaký čas. Pokud dojde k rozhodnutí využít optimalizační překladač, je vhodné to učinit hned na začátku vývoje. Později by mohly nastat problémy v nutnosti opravení některých funkcí a metod programu. Některé optimalizační metody mohou být samozřejmě provedeny i při psaní kódu. Zde je opět pár tipů, jak psát efektivnější kód.

Při psaní kódu, kde je využíváno matematických operací, je vhodné zvolit zápis „+=“ namísto zápisu „a=a+5“ viz obrázek níže (Obrázek 18). Místo dělení a násobení je možné využít bitového posunu. Posun o 1 doprava odpovídá dělení 2, naopak doleva se jedná o násobení. Pokud by bylo zapotřebí násobit například 4, bitový posun by pak byl o 2 doleva.

```
6     int a=25;
7     int b=0;
8
9     a = a + 3;
10    b = a/2;
11    a = a * 2;
12    System.out.print("A: "+a+", B: "+b);
13    /**/
14     a=25;
15     b=0;
16    a += 3;
17    b = a >> 1;
18    a = a << 1;
19    System.out.print("A: "+a+", B: "+b);
```

Obrázek 18. Java – zápis matematických operací

Dále je možné odstranit společné podvýrazy. Například když dochází ke jednomu výpočtu vícekrát viz Obrázek 19.

```
24    a = 5*b + 3;
25    b = 5*b - 4;
26    System.out.print("A: "+a+", B: "+b);
27    /*versus*/
28    int spolecnyVypocet = 5*b;
29    a = spolecnyVypocet + 3;
30    b = spolecnyVypocet - 4;
31    System.out.print("A: "+a+", B: "+b);
```

Obrázek 19. Java – odstranění společných výrazů

Je vhodné si vytvořit lokální proměnnou, než například volat v každém for cyklu funkci *array.size* apod. Při práci s řetězci, zejména pokud se často mění (přidává se k řetězci další například ve for cyklu) je lepší místo jednoduchého operátoru + nebo += společně s datovým typem String zvolit StringBuffer nebo StringBuilder, které využívají funkce jako *append()* a *toString()*. Dále je pro zvýšení výkonu aplikace vhodné používat primitivní typy místo obalů tříd – int místo Integer nebo double místo Double. Hodnota se tím ukládá na zásobník, a ne do haldy (snížení spotřeby paměti). Pokud to není nezbytně nutné nemusí docházet k využití typu BigInteger nebo BigDecimal – opět se tím sníží potřeba zabrání větší časti paměti a zvýší se například rychlost prováděných výpočtů. Dále je vhodnější pro indexové proměnné využívat typ int namísto long nebo double. Opět se tím urychlí například provádění cyklu. [28][29]

5.3 Tipy pro optimalizaci – C#

Při složitějším programování je dobré přemýšlet, jaký datový typ se zvolí pro ukládání daných hodnot. Z hlediska zlepšení výkonu kódu je schopen správně zvolený datový typ ušetřit spoustu času. Například ukládání hodnot do pole „intů“ může být přibližně 2x rychlejší než uložení stejných hodnot do proměnné typu List<int>. Při potřebě použití cyklu je vhodné dát přednost smyčce for před foreach (Obrázek 20).

```
Stopwatch stopky = new Stopwatch();
stopky.Start();
//rychlejší
for (int i = 0; i < 10000; i++)
{
    l1.Add(i);
}
stopky.Stop();

text.AppendText("For smycka cas: " + stopky.ElapsedTicks + "\n");
stopky.Restart();
//pomalejší
foreach (int i in l1)
{
    l2.Add(i);
}
stopky.Stop();
text.AppendText(" Foreach cas: " + stopky.ElapsedTicks);
```

Obrázek 20. For versus Foreach C#

V případě situace, kdy je nutné ukládat pouze malé množství dat, vychází rychlostně obecně lépe využití struktury před třídou (Obrázek 21).

```
Stopwatch sw = new Stopwatch();
sw.Start();
//rychlejsi
for (int i = 0; i < 1000; i++)
{
    struktura[i] = new Struktura();
    struktura[i].nazev = "protein 500";
    struktura[i].pocet = 2;
}
sw.Stop();
Console.WriteLine("Struktura: " + sw.ElapsedTicks);
sw.Restart();
//pomalejsi
for (int i = 0; i < 1000; i++)
{
    trida[i] = new Trida();
    trida[i].nazev = "protein 500";
    trida[i].pocet = 2;
}
sw.Stop();
Console.WriteLine("Trida: " + sw.ElapsedTicks);
```

Obrázek 21. Struktura versus třída C#

Další možností (podobně jako například u jazyka Java) je využití StringBuilderu namísto klasického stringu. To se doporučuje zejména pokud dochází k častým změnám (zřetězování, odebírání části apod.) řetězce. [31]

5.4 Výkonnostní testy

Výkonnostní testy programů jsou prováděny zejména proto, aby se otestovala požadovaná rychlost a výkonnost, která může být dána požadavkem na aplikaci. Testuje se doba odezvy spolu s propustností aplikace. Je vhodné, aby se například aplikace nenačítala příliš dlouho nebo aby zvládala požadovaný nápor uživatelů. I tyto základní věci, které by mohli být brány jako samozřejmé je potřeba otestovat a případně zjistit, v jaké části programu dochází k největším časovým režiím. S tím souvisí i zmiňovaná optimalizace, kdy poté, co jsou nalezeny části kódu, které se provádějí po delší dobu, než by bylo nutné, dochází nebo by mělo dojít

k optimalizaci alespoň těchto úseků. K hledání slabších úseků dochází při Performance profile.

Pro přesné změření výkonu testované aplikace, existují vlastnosti, které jsou brány v úvahu.

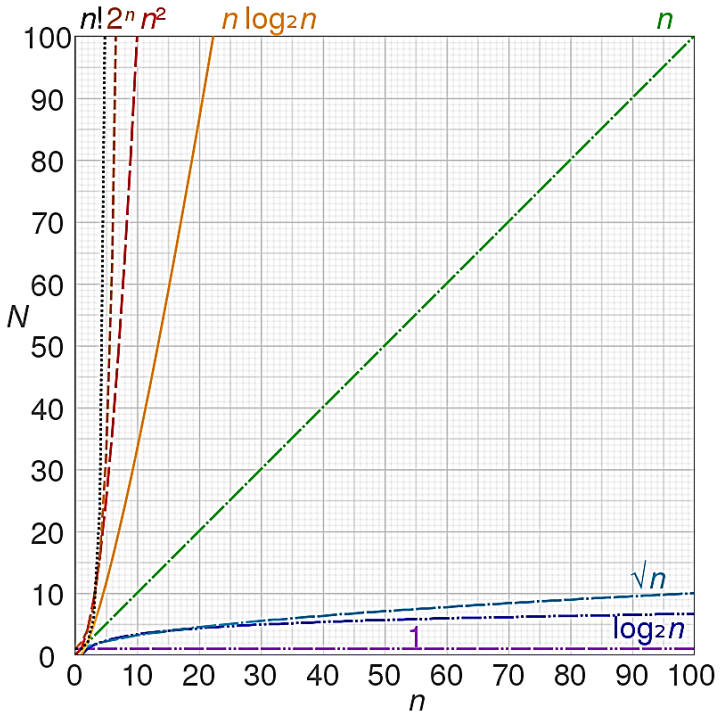
- Dostupnost – je doba, po kterou je aplikace uživateli dostupná
- Doba odezvy – je časový úsek, který je nutný k reakci programu na požadavek uživatele
- Propustnost – například kolik uživatelů je schopno být na stránce/aplikaci v jednom časovém okamžiku
- Využití – se věnuje teoretické kapacitě zdroje, jenž je využíván

Typy výkonnostních testů:

- Load testing – dochází k ověřování, zda aplikace zvládne danou zátěž a jak se chová při zátěži zvýšené
- Stress testing – slouží k nalezení horní hranice výkonu systému. Postupně zvětšovaná zátěž je přiváděna do doby, než dojde k pádu aplikace, zaplnění dostupné paměti nebo zaznamenání nějakých chyb.
- Soak testing – kontroluje stabilitu systému, odolnost a schopnost zotavení aplikace. Slouží k identifikování chyb, které se objeví až po delším čase – například pomalý únik paměti.
- Smoke testing – je zaměřeno pouze na to, kde nastala nějaká změna kódu[30]

Z oblasti výkonnostních testů se tato práce zabývá především hlediskem časové náročnosti daných algoritmů. Časová složitost souvisí s obecnějším pojmem teorie složitosti, která zkoumá, jak jsou řešitelné algoritmy náročné, a to z hlediska paměti nebo času. Právě časová složitost udává, jakým způsobem závisí počet operací během výpočtu (a potřebný čas) na množství vstupních dat. Složitost je vyjádřena pomocí funkce závisející na vstupních datech *n*. Časová složitost je oproti prostorové (potřebná paměť) o něco důležitější aspekt. Také kód se primárně optimalizuje právě na základě časové náročnosti. Díky složitosti lze rozhodnout jaký z algoritmů, které řeší stejnou věc, je na tom nejlépe. Jelikož časová složitost je závislá i na konkrétních vstupních hodnotách, které algoritmem prochází, nejen na jejich množství, lze z obecného hlediska rozlišovat náročnost v nejhorším, průměrném a nejlepším případě. V nejlepším případě například při hledání prvku v poli procházením pole for cyklem se daná hodnota nachází již na nultém indexu pole, v nejhorším se hledaná hodnota v poli nenachází. Většinou se složitost určuje pouze pro nejhorší případ. Obvykle není ani tak důležité v kolika

krocích algoritmus proběhne, ale to, jak bude počet kroků růst, když bude docházet ke zvyšování vstupních dat. V tomto případě je časová složitost označována asymptotickou notací. Pro zápis $f(x) \in O(f(x))$ platí, že funkce $f(x)$ roste nejvýše tak rychle jako funkce $g(x)$. Jedná se o neostrý horní odhad složitosti algoritmu. Například když je k dispozici algoritmus, který po přivedení 10 - ti dat na vstup trvá 10 milisekund, po přivedení 100 dat trvá 100 milisekund (zatím to vypadá dobře na lineární složitost), ale po přivedení 1000 dat na vstup trvá 10 000 milisekund (to už je horší), má algoritmus exponenciální růst. Přehled růstu počtu operací v závislosti na rostoucích vstupních datech je znázorněn na obrázku Obrázek 22. [34]



Obrázek 22. Asymptotické složitosti (N – počet operací, n – množství vstupních dat) [35]

Na základě různých řádů složitosti, které jsou vyjádřeny matematickými funkcemi, se rozlišují základní zápisy asymptotické Omikron notace (plus jejich příklady):

- O (1) – konstantní časová složitost – například funkce přehození dvou čísel
- O (log(n)) – logaritmická složitost – například binární vyhledávání
- O (n) – lineární složitost – nalezení maximální hodnoty procházením všech prvků pole pomocí for cyklu
- O (n*log(n)) – linearitmická složitost – merge sort, quicksort nebo heap sort algoritmus
- O (n^k) – polynomiální složitost, kde k je větší než 1 – bubble sort algoritmus

- O (k^n) – exponenciální složitost, kde k je konstanta – nalezeni fibonacciho posloupnosti
- O (n!) – faktoriální složitost

Upřednostňují se algoritmy, které mají lineární růst nebo jsou rychlejší. Pomalejší algoritmy se složitostí jako O(n^2) jsou nevhodné na větší objem dat. [33]

Aby bylo možné otestovat algoritmy z hlediska časové složitosti (jak dlouho trvá, než projdou data daným for cyklem a podobně), je možné využít knihoven, které obsahují funkce pro měření času.

5.4.1 Měření času Python

Zde se nabízí využití knihovny *datetime*, jejíž použití je znázorněno na obrázku Obrázek 23. Testuje se, jak dlouho trvá funkce *Forsmycka()*, která prochází for cyklus a vypisuje čísla od 0 po 99. Pro toto testování je vhodné zapisovat do tabulky více naměřených hodnot (spustit program vícekrát) a poté z nich vypočítat průměrnou dobu trvání funkce.

```
from datetime import datetime

def Forsmycka():
    for i in range(100):
        print(i)
start_time = datetime.now()
Forsmycka()
time_elapsed = datetime.now() - start_time
print('For (hh:mm:ss.ms) {}'.format(time_elapsed))
```

Obrázek 23. Knihovna datetime Python

Další možností je využití knihovny *timeit* a její funkce *default_timer()*. Ukázka je zobrazena obrázkem Obrázek 24, kdy je vrácen rozdíl času konce a začátku prováděné funkce. Tentokrát nemusí docházet k formátování času, je rovnou zobrazen ve vhodném formátu. Opět by měl být nalezen průměr několika vrácených časových hodnot.

```
from timeit import default_timer

def Forsmycka():
    for i in range(100):
        print(i)
start=default_timer()
Forsmycka()
end=default_timer()
print(end-start)
```

Obrázek 24. Knihovna timeit Python funkce default_timer

Asi nejvhodnější metodou, jak změřit čas prováděné funkce, je použít rovnou metodu *timeit()* ze zabudované knihovny *timeit*. V tomto případě je již vrácena průměrná doba provádění daného úseku programu, jelikož funkce *timeit()* spouští daný úsek programu asi milionkrát (defaultně). Funkce *timeit.timeit(stmt, setup, timer, number)* přijímá 4 argumenty:

- Stmt – zde se nastavuje kód, u kterého je požadováno změření doby jeho provádění
- Setup – případné nastavení – kód, který je spuštěn před funkcí v stmt. Například importování knihoven, funkce volané v průběhu stmt a podobně.
- Timer – hodnota časovače, u timeit se obvykle nenastavuje
- Number – funkce v stmt je spuštěna tolikrát, jaké je číslo v parametru Number (defaultní hodnota je 1 000 000) [36]

Na obrázku Obrázek 25 je uvedena ukázka změření průměrné doby trvání z 10 spuštění funkce Forsmycka použitím funkce *timeit()*.

```
import timeit

def Forsmycka():
    for i in range(100):
        print(i)
cas = timeit.timeit('Forsmycka()',setup='from __main__ import Forsmycka ; import timeit',number = 10)
print(cas)
```

Obrázek 25. Funkce timeit.timeit() Python

5.4.2 Měření času C#

V C# se pro přesné zjištění doby, po kterou běží určitá část kódu, nabízí využití třídy *Stopwatch* z namespace *System.Diagnostics*. Nejprve je vytvořena instance třídy Stopwatch, poté je volána metoda *Start()*, která zahájí měření času. Dále se vykoná část programu, kterou je zapotřebí změřit, dojde k ukončení měření času pomocí metody *Stop()* a nakonec je zjištěn uplynulý čas trvání měřené části kódu pomocí vlastnosti Elapsed. Ukázka měření času je uvedena na obrázku Obrázek 26. V tomto případě je opět měřen čas trvání cyklu for, který vypisuje prvky od 0 po číslo 99. [37]

```
Stopwatch stopwatch = new Stopwatch();
stopwatch.Start();
for (int i = 0; i < 100; i++) {
    Console.WriteLine(i);
}
stopwatch.Stop();
Console.WriteLine(stopwatch.Elapsed);
```

Obrázek 26. Ukázka třídy Stopwatch C#

Pro přesnější změření je vhodné zjistit průměrný čas například ze 100 provádění cyklu (viz Obrázek 27).

```
Stopwatch stopwatch = new Stopwatch();
int pocetCykluFor = 100;
stopwatch.Start();
for (int i = 0; i < pocetCykluFor; i++) {
    merenaForSmycka();
}
stopwatch.Stop();
Console.WriteLine("Cas"+stopwatch.Elapsed.TotalMilliseconds/10/pocetCykluFor);
```

Obrázek 27. Průměr ze 100 prováděných cyklů for C#

5.4.3 Měření času Java

Zde je vhodným zástupcem funkce *System.nanoTime()*, která vrací hodnotu aktuálního času v nanosekundách. Při měření uplynulého času při běhu části programu se odečítají dvě časové hodnoty, a to konec a začátek dané části kódu (viz Obrázek 28).

```
long start = System.nanoTime();
for(int i = 0; i<100;i++){
System.out.print(i+"\n");
}
long konec = System.nanoTime();
System.out.print("cas: "+(konec-start));
```

Obrázek 28. funkce nanoTime() Java

Další možností je využití funkce *System.currentTimeMillis()*. Tato funkce vrací hodnotu v milisekundách. Problém však je, že někdy nemusí být přesná, proto se častěji využívá metoda *nanoTime()* zmíněná výše. Požívá se podobně jako *System.nanoTime()*. [38]

Další možností je importování API *com.google.common.base* a třídy *Stopwatch*. Oproti *System.nanoTime()* vrací pouze relativní hodnotu uplynulého času, a to pomocí metody *Stopwatch.elapsed()*. Při použití třídy *Stopwatch* se využijí její metody jako *Stopwatch.createStarted()*, což vytvoří a zapne stopování času, nebo *Stopwatch.stop()*, což stopky naopak zastaví (viz Obrázek 29). [39]

```
Stopwatch stopwatch = Stopwatch.createStarted();
for(int i = 0; i<100;i++){
System.out.print(i+"\n");
}
stopwatch.stop();
System.out.print("cas stopWatch: "+stopwatch.elapsed(MILLISECONDS));
```

Obrázek 29. Stopwatch Java

II. PRAKTICKÁ ČÁST

6 TVORBA GUI

Pro implementaci šifer jsou zvoleny jazyky Python, C# a Java. Tyto jazyky jsou vybrány zejména z důvodu jejich častého výskytu v kurzech vyučovaných na fakultě.

Přehled vývojových prostředí pro implementaci šifer a tvorbu GUI je znázorněn tabulkou Tabulka 23.

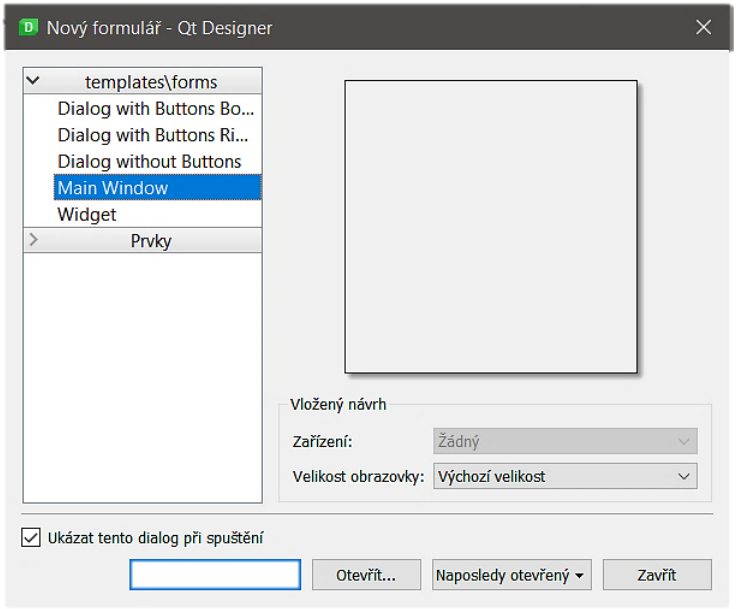
Tabulka 23. Přehled vývojových prostředí pro implementaci šifer a tvorbu GUI

	Python	**C#**	**Java**
Implementace kódu	Spyder 4.1.4.	Visual Studio 2019	Apache Netbeans IDE 12.2
Tvorba GUI	Qt Designer 5.15.1	Visual Studio 2019	JavaFX SceneBuilder
Dostupnost vývojových prostředí na operačním systému	Windows, MacOS, Linux	Windows, MacOS	Windows, MacOS, Linux

Pro implementaci kódu v jazyce Python je zvoleno vývojové prostředí Spyder 4.1.4. Tvorba uživatelského rozhraní probíhá v prostředí Qt Designer 5.15.1. Pro jazyk C# je zvoleno Visual Studio 2019, a to přímo Aplikace Windows Forms (.NET Framework), u které je možné tvořit GUI přímo ve vývojovém prostředí spolu s kódem funkcionality. V Javě je využito vývojového prostředí Apache NetBeans IDE 12.2. Tvorba GUI zde probíhá pomocí JavaFX SceneBuilderu, který umožňuje vizualizaci a změnu elementů GUI během návrhu, což je výhodnější než vytváření GUI pouze pomocí kódu.

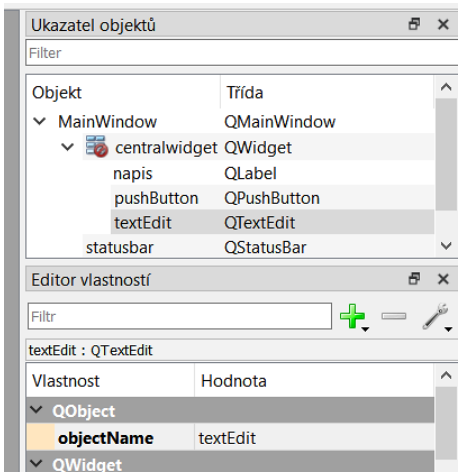
6.1 Python

Jak již bylo zmíněno výše, tvorba GUI probíhá v prostředí Qt Designer. Při tvorbě formuláře pro zadávání klíčů a textu k šifrování či dešifrování je nejprve zvoleno vytvoření nového okna dle předlohy (viz Obrázek 30). Tento soubor je uložen tam, kde bude později vytvořen soubor s python kódem pro dané GUI.



Obrázek 30. Tvorba GUI Python

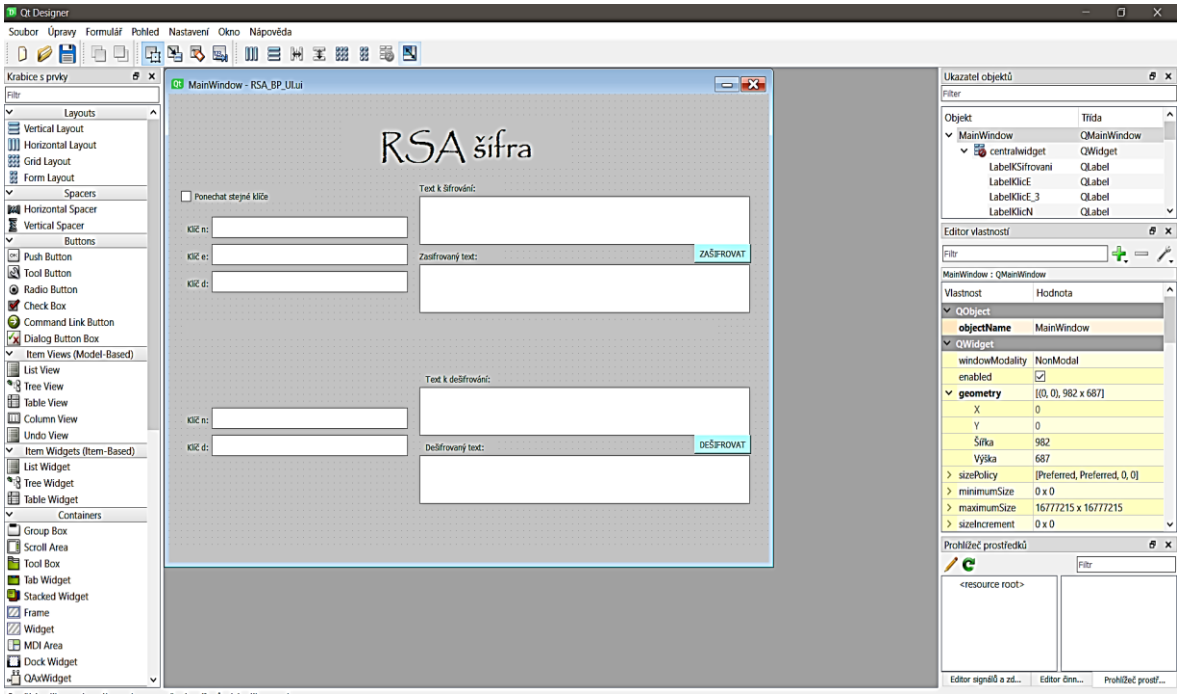
Výhodou je, že může být nastavena i velikost okna formuláře, které se pak přímo zobrazuje i uživateli. Z nabídky prvků je nejvíce využit Label, TextEdit a PushButton. Tyto prvky mohou být jednoduše přetáhnuty do hlavního okna formuláře. Ovládání je zde velmi intuitivní a může vzniknout pěkně vypadající formulář, který bude uživateli dostupný. Je dobré si dát pozor na pojmenování prvků, ke kterým je přistupováno ze souboru, kde je napsán zdrojový kód. Přejmenování objektů probíhá jednoduchým dvojklikem na název v oblasti Ukazatel objektů nebo v Editoru vlastností (viz Obrázek 31).



Obrázek 31. Ukazatel objektů a Editor vlastností Ot Designer

Dále je zde možnost nastavení vlastností jako přesné umístění objektu pomocí souřadnic, velikost objektu, nastavení písma nebo například pozadí, a to pomocí vlastnosti styleSheet

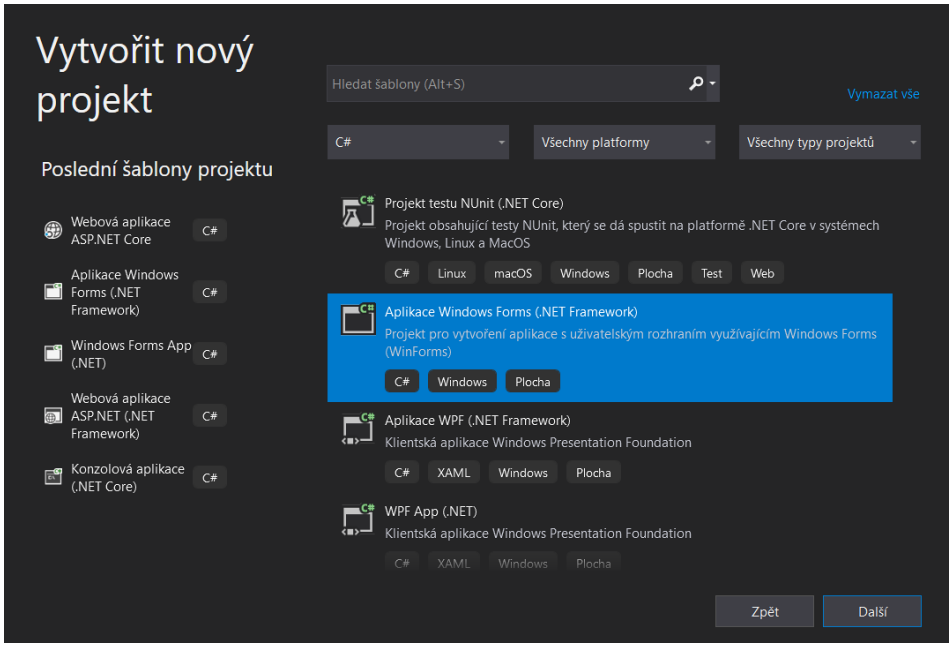
(zde musí být vloženo background: barva). Vlastnost Scrollbar je defaultně nastavena, takže není nutné si zde dávat takový pozor, jako například u Windows Forms níže. Výsledkem může být základní GUI pro šifrování jako na Obrázek 32. Aby mohl být soubor s GUI propojen s kódem Pythonu, je potřeba do následně vytvořeného python souboru vložit část kódu, který je popsán v kapitole 7.1.1.



Obrázek 32. Ukázka GUI Qt Designer

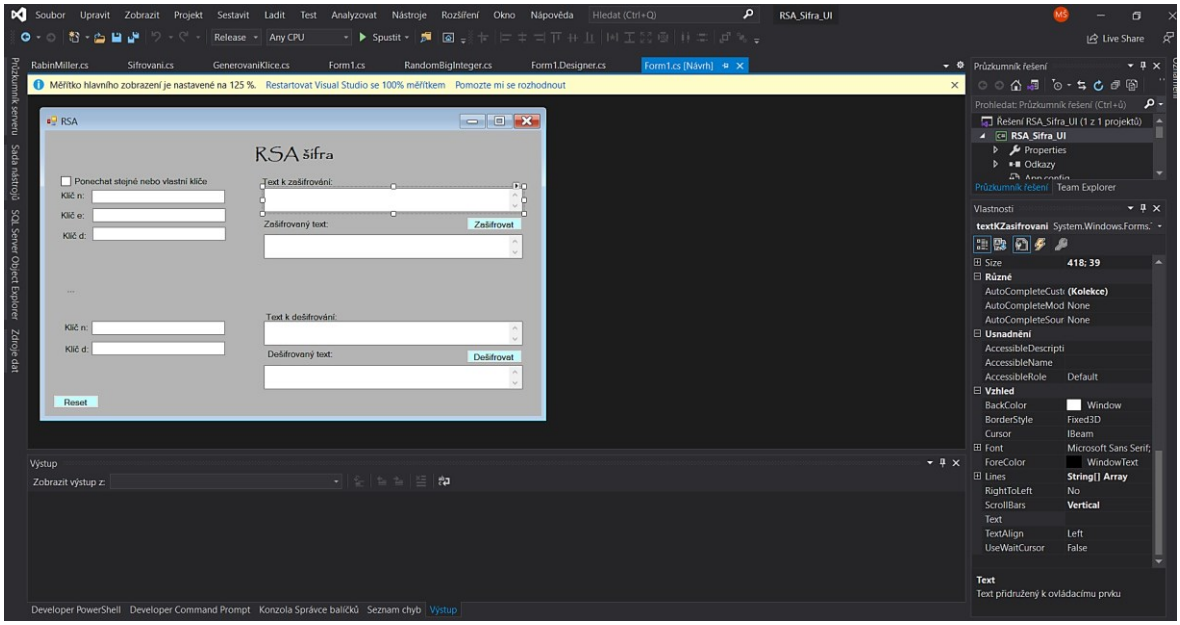
6.2 C#

Jelikož Windows Forms poskytují přímo ve Visual Studiu tvorbu GUI, nemusí docházet k využití jiného prostředí, což je obecně považováno za výhodu. Hned na začátku tvorby projektu je tedy zvolena Aplikace Windows Forms (.NET Framework). Volbu, k vytvoření projektu, lze vidět na Obrázek 33.



Obrázek 33. Ukázka vytvoření projektu Windows Forms (Visual Studio)

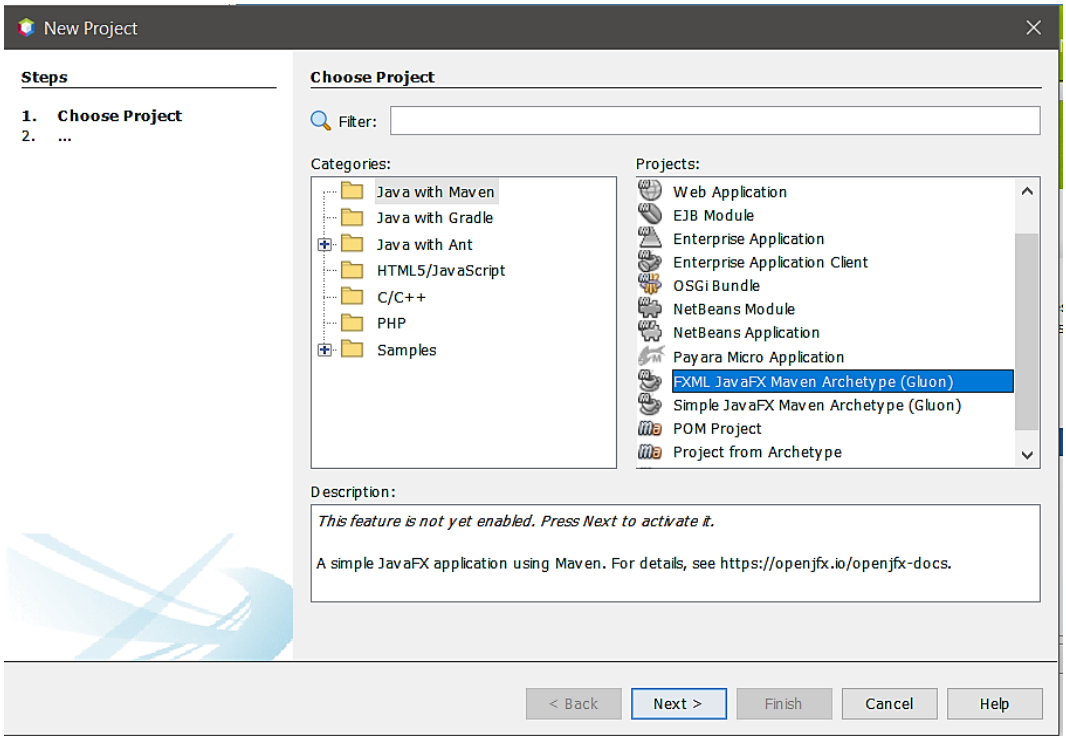
Prostředí hned automaticky vytvoří základní formulář, kde může dojít k přetáhnutí objektů ze sady nástrojů podobně jako v Qt Designer. Opět je zde možnost úpravy vlastností jako velikost, uspořádání objektů pomocí souřadnic, jméno objektu apod. To vše je nastavováno v oblasti Vlastnosti. Toto uspořádání působí o něco méně přehledně jako Qt Designer výše. Trošku nemilé je zde vkládání textu do objektů (např. tvorba nadpisů). Není možné tuto vlastnost upravit pouze dvojklikem přímo na objekt ve formuláři, jak by se pravděpodobně očekávalo. To je pak vytvořena přímo metoda Prvek_Click, která reaguje například na kliknutí na tlačítko. Na druhou stranu je to i výhoda – metoda se generuje automaticky, může se do ní rovnou vkládat kód zajišťující například šifrování. Text ve formuláři musí být do objektu vložen pomocí vlastnosti Text. Nevýhodou oproti Qt Designeru je, že pokud je zapotřebí víceřádkový TextBox, musí dojít k manuálnímu nastavení vlastnosti MultiLine; toto se tedy neděje automaticky během vertikálního zvětšení objektu. Je také většinou vhodné aktivovat vlastnost ScrollBars, která je defaultně nastavena na None. Naopak malou výhodou je nastavení barvy pozadí, což přímo umožňuje vlastnost BackColor. Výstupem pak může být formulář pro šifrování jako na obrázku Obrázek 34. Vzhledově působí vytvořený formulář téměř totožně, jako při tvorbě GUI v Qt Designeru. V kódu funkcí se pak přistupuje k prvkům uživatelského rozhraní pomocí názvů, které se nastaví při tvorbě formuláře.



Obrázek 34. Ukázka GUI v prostředí Visual Studio

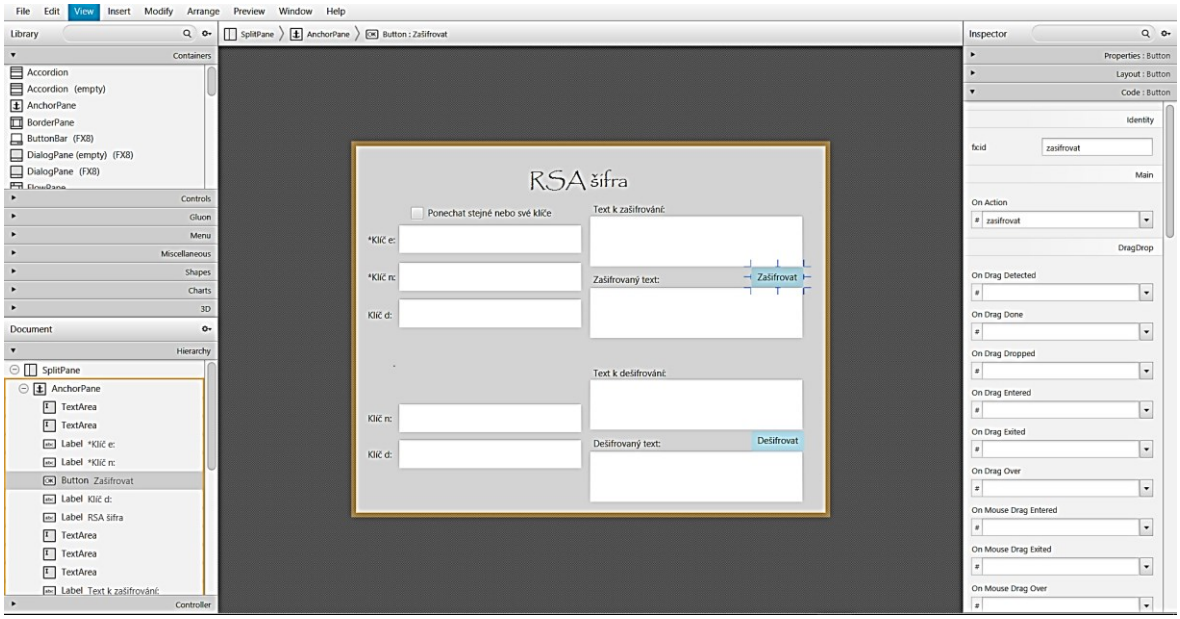
6.3 Java

Pro tvorbu uživatelského grafického rozhraní v Javě byl zvolen JavaFX. Má velkou výhodu v tom, že podobně jako v předcházejících jazycích může být využito grafického designeru, v tomto případě JavaFX Scene Builder, a nemusí být psán přímo kód FXML. Jedná se podobně jako u Qt Designeru o samostatnou aplikaci, díky které je usnadněn návrh rozložení a tvorby jednotlivých objektů. Po vytvoření projektu v NetBeans – FXML JavaFX Maven Archetype (viz Obrázek 35) se již automaticky vytvoří základní kontrolery a soubory fxml.



Obrázek 35. Vytvoření projektu v Apache NetBeans

Po otevření fxml souborů z projektu v SceneBuilder designeru je zde opět možnost přetažení jednotlivých prvků. Zde se již samotný základ formuláře negeneruje automaticky, je vybírán z knihovny objektů jako ostatní prvky, což může být z počátku matoucí. Při tvorbě GUI zde byl menší problém při určení velikosti formuláře. Pokud se nastavil vetší, nedocházelo k jeho plnému zobrazení, jako tomu je u výše popsaných možností tvorby GUI u jazyků C# a Python. Opět je zde na výběr nastavení různých vlastností, od velikosti, písma, stylu pozadí, zarovnání objektů podle souřadnic až po nastavení funkcí z kódu, které budou například při stisknutí tlačítka probíhat. Toto je možné chápat jako výhodu tohoto prostředí, jelikož se toto propojení s funkcí v odpovídajícím kontroleru nemusí již nastavovat pomocí kódu. Musí se dávat pozor, zda je v oblasti Controler nastaven ten, ve kterém se s daným souborem layoutu pracuje. Místo názvu se u každého objektu nastavuje ID, které musí souhlasit s použitím v kódu java souborů. Pokud se v kódu odpovídajícího kontroleru definují funkce a proměnné, které budou odpovídat jednotlivým prvkům formuláře, stačí je ve SceneBuilderu pouze k daným prvkům vybrat. Vlastnost ScrollBar u TextArea je nastavena automaticky. GUI ve SceneBuilderu může vypadat jako na Obrázek 36. Samotný SceneBuilder je řešený přehledně a nastavení vlastností či výběr objektů je uživatelsky přívětivé.



Obrázek 36. Ukázka GUI vytvořeného v JavaFX SceneBuilderu

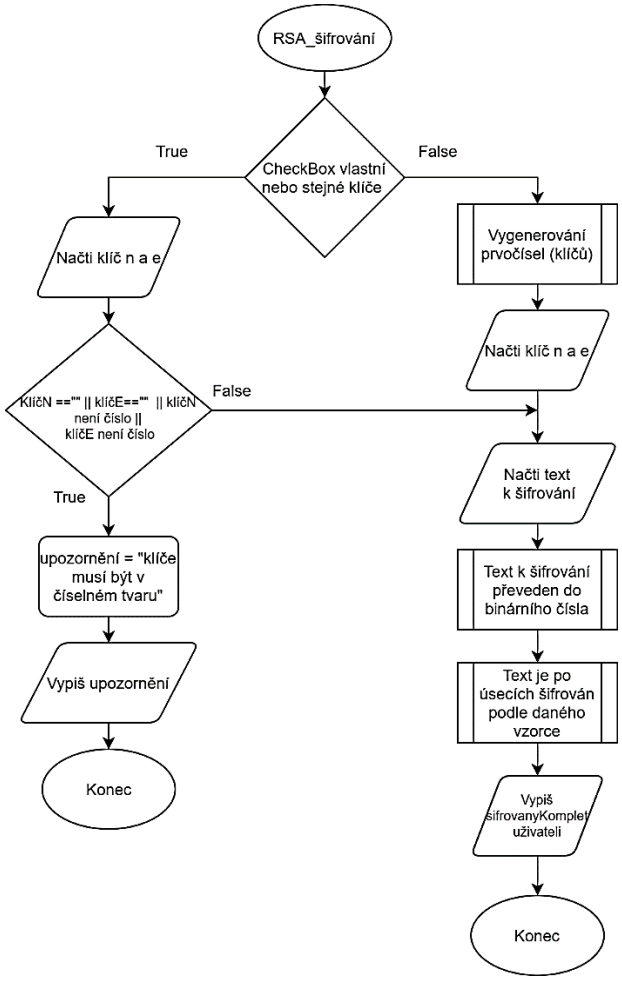
7 IMPLEMENTACE ŠIFER

V této kapitole jsou popsány implementace šifer RSA, Zubatka a One – time Pad v jednotlivých jazycích a je zdůvodněno, proč došlo k výběru právě daných šifer. Dále jsou zde popsány výhody a nevýhody použití jazyků, ukázky kódu a vývojových diagramů.

7.1 RSA

RSA šifra je vybrána z důvodu, že tvoří klasického zástupce asymetrické kryptografie a využívá složitých matematických operací, což může být v některých jazycích komplikací.

Šifrování pomocí RSA využívá generování a součinu prvočísel. Princip je již popsán výše v kapitole 4.3. Struktura programu je pro přehlednost popsána prostřednictvím vývojových diagramů, které popisují šifrování a dešifrování. Základní části diagramů lze vidět níže. Diagramy podprogramů, které jsou znázorněny pouze názvem vykonávané funkce jsou doloženy v příloze P I.

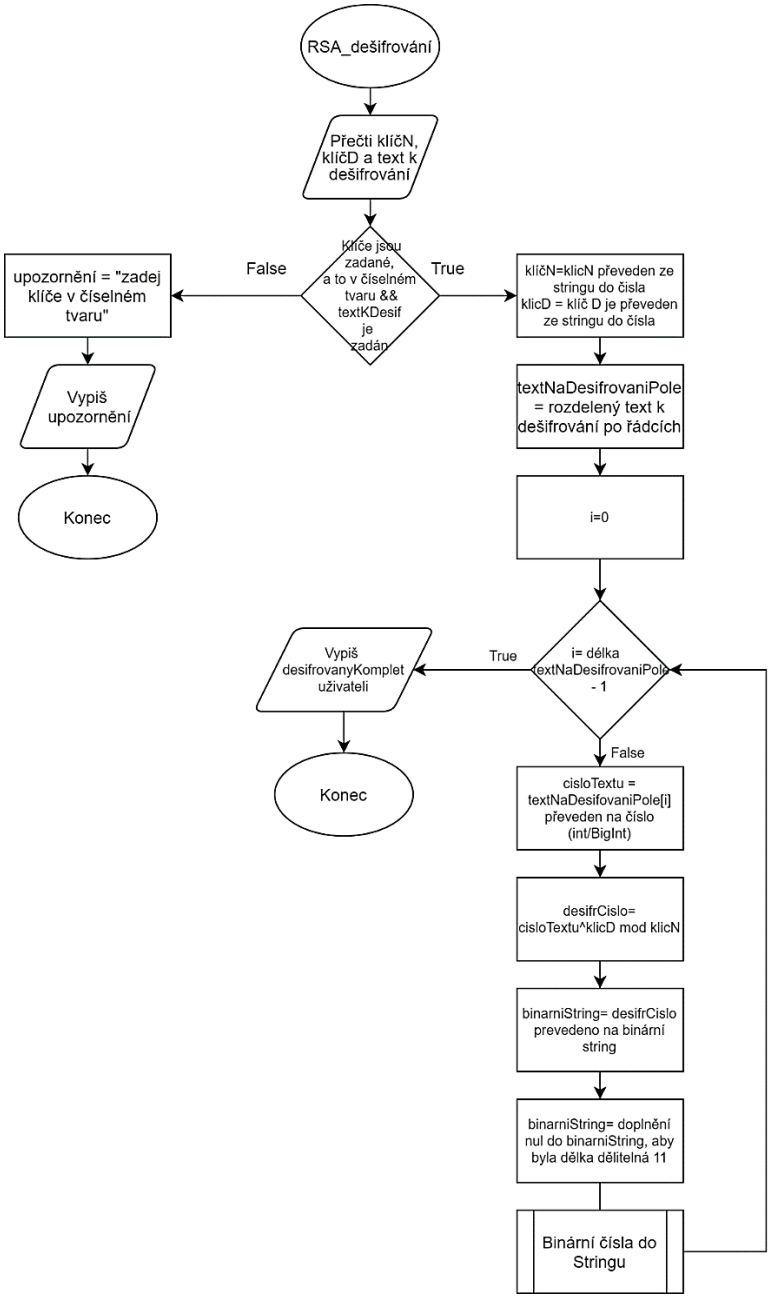


Obrázek 37. Vývojový diagram šifrování RSA

Na diagramu průběhu šifrování RSA (Obrázek 37) lze vidět, že nejprve dochází k ověření, zda uživatel chce použít stejné klíče nebo chce vygenerovat klíče nové (v kódu řešeno pomocí check – boxu). Pokud zvolí generování nových klíčů, je volána funkce určena na generování klíčů, ve které jsou nejprve vygenerovány náhodná prvočísla, poté je vypočten klíč *n* jako součin těchto prvočísel a vypsán uživateli do formuláře. Následně dochází k výpočtu čísla *Fi*. Dále je získáváno číslo *e* jako náhodné číslo z intervalu 1 až Fi-1. Získávání čísla *e* probíhá do té doby, až je splněna podmínka *gcd(e,Fi)* je rovno 1. Číslo *e* je vypsáno uživateli. Nakonec dochází k výpočtu a zobrazení čísla *d*. Klíč *n* spolu s klíčem *e* jsou dále využity k šifrování textu. Pokud však uživatel zvolí, že si přeje ponechat stejné klíče (v případě, že byly před tím vygenerovány a vypsány) nebo chce zadat vlastní, není volána funkce generování klíčů, ale jsou přímo načteny klíče od uživatele. Pokud uživatel klikne na tlačítko šifrovat, má zvolenou možnost vlastních klíčů, ale zapomene je zadat, bude na tuto skutečnost upozorněn a metoda šifrování se ukončí.

Po správném načtení klíčů je načten text od uživatele určený k šifrování. Tento text je po znacích převeden do binárního čísla (každý znak odpovídá binárnímu číslu s délkou 11 a tyto binární čísla jsou spojovány do jednoho). Dále je toto velké binární číslo rozděleno po úsecích s délkou 88. Každý úsek je převeden do dekadické podoby, šifrován pomocí vzorce a přidán do šifrovaného kompletu. Po zašifrování všech úseků je šifrovaný komplet vypsán uživateli.

Dešifrování je znázorněno na diagramu níže (Obrázek 38). Nejprve jsou přečteny klíče *n* a *d* spolu s textem k dešifrování zadané uživatelem. Následně je ověřováno, zda klíče a text nebyly prázdné a obsahovaly pouze čísla. Pokud tomu tak není, uživatel je upozorněn a funkce dešifrovaní je ukončena. Pokud je přečtení vstupů úspěšné, dochází k převedení klíčů z řetězce znaků do čísel. Následně je text k dešifrování rozdělen po řádcích do pole. Dále je každá položka pole textu na dešifrování převedena do dekadického čísla (původně se jedná o řetězec čísel), dešifrována pomocí vzorce a převedena na binární číslo, které je doplněno nulami tak, aby jeho délka byla dělitelná 11. Toto binární číslo je po úsecích s délkou 11 převedeno na znaky, které se ukládají do dešifrovaného kompletu. Po průchodu celého pole textu na dešifrování je dešifrovaný komplet vypsán uživateli.



Obrázek 38. Vývojový diagram dešifrování RSA

7.1.1 Python

Po otevření vývojového prostředí Spyder se nejprve zvolí File – New file. Aby došlo ke spárování python souboru s GUI vytvořeným pomocí Qt Designeru, je potřeba tento soubor uložit do složky, ve které se nachází již zmiňovaný soubor s UI. Pro propojení je zapotřebí do python souboru vepsat kód, který lze vidět níže (Obrázek 39).

```
8
9   import sys
10  from PyQt5.QtWidgets import QApplication, QMainWindow
11  from PyQt5 import QtGui, uic
12
13  qtCreatorFile = "nazevSouborusGUI.ui"
14  Ui_MainWindow, QtBaseClass = uic.loadUiType(qtCreatorFile)
15
16  class MyApp(QMainWindow, Ui_MainWindow):
17
18      #zde vkladat funkce kodu
19
20
21      def __init__(self):
22          QMainWindow.__init__(self)
23          Ui_MainWindow.__init__(self)
24          self.setupUi(self)
25          #spojeni tlacitek v GUI s pozadovanymi funkcemi z kodu
26          self.pushButtonZasifrovat.clicked.connect(self.Zasifruj)
27          self.pushButtonDesifrovat.clicked.connect(self.Desifruj)
28
29
30  if __name__ == "__main__":
31      app = QApplication(sys.argv)
32      window = MyApp()
33      window.show()
34      sys.exit(app.exec_())
```

Obrázek 39. Ukázka kódu pro propojení .py souboru s GUI

Řádek 9-11: importování potřebných knihoven

Řádek 13: určeno GUI, které se má volat při spuštění programu (podle jeho názvu)

Řádek 18: dovnitř třídy *MyApp* se vkládá další kód programu (například funkce pro šifrování a dešifrování)

Řádek 26: *pushButtonZasifrovat* – název prvku tlačítka v GUI, z jehož pomocí se má spustit funkce *Zasifruj* z python souboru, která bude později implementována

Řádek 27: *pushButtonDesifrovat* – název prvku tlačítka v GUI, z jehož pomocí se má spustit funkce *Desifruj* z python souboru (pojmenování závisí samozřejmě na každém, ale musí sedět jak v GUI, tak zde)

Tento postup vytvoření python souboru a propojení s GUI je totožný i u ostatních šifer.

Jelikož je postup šifrování i dešifrování již znázorněn pomocí vývojových diagramů, budou zde popsány pouze některé části (i z hlediska možnosti porovnání daných jazyků).

Aby mohlo dojít ke generování náhodného čísla nebo prvočísla a použití matematických operací, jsou v tomto případě importovány knihovny, které lze vidět na obrázku Obrázek 40 i s knihovnami potřebnými k propojení s GUI nebo měření času.

Knihovna *random* a *symphy – randprime* slouží ke generování náhodného čísla a prvočísla při tvorbě klíčů, knihovna *math* je zde z důvodu využití funkce *math.gcd(),* která je taktéž

využita při tvorbě klíčů. Knihovna *re* slouží svou metodou *re.findall()*, která je využita při rozdělování čísla po daných úsecích.

```
8
9   from PyQt5 import QtWidgets
10  import sys
11  from PyQt5.QtWidgets import QApplication, QMainWindow
12  from PyQt5 import QtGui, uic
13  import random
14  import math
15  from sympy import randprime
16  import re
17  from time import time
```

Obrázek 40. Importované knihovny u šifry RSA (Python)

Samotné šifrování tedy začíná zjištěním, zda uživatel zvolil možnost ponechání (případně své zadání) klíčů. V Pythonu je ověřen checkBox, případně klíč viz Obrázek 41.

```
72          #overi zda uzivatel zaskrl checkBox a chce ponechat stejne nebo zadat vlastni klice
73          if(self.checkBoxVlastniKlice.isChecked() is True):
74              strKlicE=self.textEditE.toPlainText()
75              strKlicN=self.textEditN.toPlainText()
76              #overi zda jsou klice zadany a jsou v ciselnem tvaru
77              if(strKlicE == '' or strKlicN == '' or strKlicN.isdigit()==False or strKlicE.isdigit()==False):
78                  self.Upozorneni.setText('Prosím zadejte klíče e a n určené pro šifrování (v číselném tvaru).')
79                  return False
80          else:
81              #pokud uzivatel chce vytvorit nove klice, je volana funkce pro jejich vygenerovani
82              self.VytvorKlice()
```

Obrázek 41. Ověření CheckBoxu – Python

Text k zašifrování je získán zápisem: *otevrenyTextIn=(self.textEditKSifrovani.toPlain-Text())* – k prvkům z GUI se vždy přistupuje přes *self.nazevPrvku*.

Převod textu do binární podoby může probíhat způsobem, znázorněným na Obrázek 42.

```
86          binarniForma=""
87          #projde se kazdy znak v otevrenem textu
88          for i in otevrenyTextIn:
89              a=ord(i)#prevede se do ascii cisla
90              b="{0:b}".format(a)#prevede se do binarniho cisla
91              bi=self.vlozNuly(b)#vlozi se 0 pred cislo, dokud je mensi nez 11 cislic
92              binarniForma+=bi#jednotlive binarni cisla se spoji do jednoho
```

Obrázek 42. Převod textu do binární podoby (Python)

Pro rozdělení textu v binární podobě po úsecích s délkou 88 nemusí procházet například for cyklem. V Pythonu se nabízí knihovna *re* se svou metodou *findall()* a toto rozdělování se díky tomu poněkud usnadní (na rozdíl od C# nebo Javy). Rozdělení textu po úsecích a následný převod do dekadických čísel, zašifrování a doplnění do kompletního zašifrovaného textu je lze zhlédnout na Obrázek 43.

```
104         rozdelene=re.findall('.{1,88}',binarniForma) #rozdeli celkove binarni cislo po 88cich
105
106         #sifrovanyKomplet pro ulozeni celkoveho zasifrovaneho textu
107         sifrovanyKomplet=[]
108         #projdou se vsechny binarni cisla s delkou 88
109         for i in rozdelene:
110              asc=int(i, 2)#vetsinou po 88cich se prevede do ascii cisla
111              st=pow(asc,e,n)#ascii cislo sifrovane podle vzorce OT^e mod n
112              sifrovanyKomplet.append(st)#zasifrovana cast textu je pridana do sifrovaneho kompletu
113         #obsah pole sifrovaneho kompletu je spojen pomoci \n a nasledne vypsan uzivateli
114         sifrovanyKomplet=('\n'.join(map(str, sifrovanyKomplet)))
115         self.textEditSifrovanyText.setText(str(sifrovanyKomplet))
```

Obrázek 43. Ukázka části šifrování RSA (Python)

Funkce, která při šifrování generuje klíče je znázorněna na Obrázek 44.

```
36      def VytvorKlice(self):
37          start=time()
38          #promenne a, b urcujici naslednou delku prvocisel p a q
39          a=10**15
40          b=10**16
41          #vygeneruje nahodne prvocislo v intervalu (a,b)..ne vcetne b
42          p=randprime(a,b)
43          q=randprime(a,b)
44          #klic n
45          n=p*q
46          #vypise klic N do formulare
47          self.textEditN.setText(str(n))
48          #nastavi cislo Fi podle vzorce
49          Fi=(p-1)*(q-1)
50
51          #cislo e musi byt v danem intervalu a nesmi byt soudelne s cislem Fi, coz zajistuje nasledujici while
52          spatneE=True
53          while spatneE:
54          #vygeneruje se nahodne cislo v intervalu 1 az Fi-1, u kterého je overena nesoudelnost s Fi pomoci math.gcd
55              e=random.randrange(1, Fi)
56              if (math.gcd(e,Fi) !=1):
57                  spatneE=True
58              else:
59                  #pokud cislo e je nesoudelne s cislem Fi, je vypsano uzivateli a smycka while dale neprobiha
60                  self.textEditE.setText(str(e))
61                  spatneE=False
62          #klicova cast d odpovida cislu e^-1 mod Fi
63          d=self.modinv(e,Fi)
64          #d je zobrazeno uzivateli
65          self.textEditD.setText(str(d))
66          konec=time()-start
67          print("cas RSA VytvorKlice:"+str(konec))
68
```

Obrázek 44. Generování klíčů RSA (Python)

Řádek 37, 66 a 67: slouží na měření doby trvání – pro samotné šifrování nejsou podstatné

Řádek 42 a 43: generování náhodných prvočísel *p* a *q*. Zde je velkou výhodou, že se opravdu jedná o prvočísla a nemusí být nijak složitě ověřovány jako v C# nebo Javě.

Řádek 45: z prvočísel je pomocí jejich součinu vypočten klíč *n*

Řádek 47: klíč *n* je zobrazen uživateli do určeného pole v grafické rozhraní (přístup k jednotlivým prvkům pomocí *self.nazevPrvku*)

Po nastavení čísla *Fi* (řádek 49) se od řádku 52 až 61 generuje náhodné číslo *e* pomocí smyčky while – zde je využito knihovny *random* a dále je ověřováno, zda je nesoudělné s číslem *Fi*. To je určeno pomocí knihovny a funkce *math.gcd()* – pokud jsou čísla

nesoudělná – vrací se hodnota 1, hodnota klíče *e* je vypsána uživateli (řádek 60) a smyčka je ukončena.

Řádek 63: je zde vypočten klíč *d*. Zde je naopak jedna nevýhoda oproti ostatním zmiňovaným jazykům, jelikož pro výpočet inverzního modula není dána metoda knihovny – musí se funkce napsat ručně (viz Obrázek 45).

```
23      #SO
24      def egcd(self,e, Fi):
25          if e == 0:
26              return (Fi, 0, 1)
27          g, y, x = self.egcd(Fi%e,e)
28          return (g, x - (Fi//e) * y, y)
29      #SO
30      def modinv(self,e, Fi):
31          g, x, y = self.egcd(e, Fi)
32          if g != 1:
33              raise Exception('Není inverzni modulo')
34          return x%Fi
```

Obrázek 45. Výpočet inverzního modula (Python)

Funkce dešifrování může být naimplementována například jako na Obrázek 46. Jednotlivé kroky funkce jsou popsány pomocí komentářů v kódu. Velkou výhodou Pythonu je, že nemusí docházet k zadání datového typu u proměnných. Člověk se nemusí zabývat tím, jestli postačí zvolit int nebo musí dojít k použití jiného datového typu, jako u ostatních jazyků. Mělo by být také oceněno množství dostupných užívaných funkcí – například jednoduchý převod na int, string nebo char (řádek 159 a 182). Funkce *time()* opět slouží pouze k měření času a pro funkcionalitu dešifrování není potřebná.

```python
def Desifruj(self):
    start=time()
    self.Upozorneni.setText('')
    strKlicN = self.textEditNDesifr.toPlainText()
    strKlicD = self.textEditDDesifr.toPlainText()
    #ulozeni zadaneho textu na desifrovani
    textNaDesifrovani=self.textEditKDesifrovani.toPlainText()
    #overeni zda jsou zadane klice pro desifrovani v ciselnem tvaru a text pro desifrovani
    if(strKlicN == '' or strKlicD == '' or strKlicN.isdigit()==False or strKlicD.isdigit()==False or textNaDesifrovani =='' ):
            self.Upozorneni.setText('Prosím zadejte klíče a text určené pro dešifrování! (v číselném tvaru)')
            return False
    #klice n a d jsou prevedeny na cislo
    n=int(strKlicN)
    d=int(strKlicD)
    #text na desifrovani je rozdelen do listu podle konce radku
    textNaDesifrovani=self.StringDoListu(textNaDesifrovani)

    startUlozeniDesifrText=time()
    #promenna na ulozeni celkoveho desifrovaneho textu
    text=""
    #prochazeni kazdeho prvku v listu
    for i in textNaDesifrovani:
        #overeni zda text pro desifrovani obsahuje pouze cisla
        try:
            i=int(i) #ciselny string je preveden na cislo
        except:
            self.Upozorneni.setText('Prosím zadejte text pro dešifrování ve tvaru čísel.')
            return False
        desifr=pow(i,d,n) #cislo je desifrovano pomoci vzorce ST^d mod n
        b="{0:b}".format(desifr)#desifrovana cast je prevedena do binarniho cisla
        bO=self.vlozNuly(b) #doplní se do velkeho binarniha cisla 0 na zacatek, tak aby bylo delitelne 11
        rozdelenePo11=re.findall('.{1,11}',bO) #binarni cislo je rozdeleno po delce 11
        #kazda binarni cast o delce11 je prevedena zpet na asci cislo a na znak
        for i in rozdelenePo11:
            asc=int(i, 2)
            znak=chr(asc)
            #desifrovany znak je ulozen do desifrovaneho kompletu
            text+=znak
    #desifrovany text je zobrazen uzivateli
    self.textEditDesifrovany.setText(text)
```

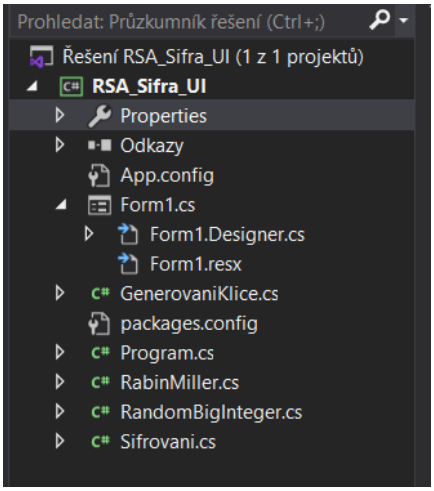
Obrázek 46. Funkce dešifrování RSA (Python)

S Pythonem se pracuje velmi instinktivně a nabízí velmi široké spektrum metod a knihoven, které jsou zde velmi nápomocné – zejména při generování klíčů. Z počátku může trvat, než si člověk zvykne na novou syntaxi kódu, která se ovšem zdá být přehlednější a jednodušší na zápis než například C#. Nemusí docházet k psaní středníků na konci řádku, složených závorek a v rámci optimalizace lze velmi redukovat řádky.

7.1.2 C#

V C# je využito prostředí Visual Studia pro tvorbu GUI i přímo programu, jak již bylo zmíněno. Po vytvoření projektu a formuláře (viz kapitola 6.2) se dvojitým kliknutím na prvek ve formuláři lze dostat ze souboru Form1.cs [návrh] do souboru Form1.cs, kde je rovnou vytvořena funkce, která bude reagovat na práci s daným prvkem. Například pří dvojitém kliknutí na tlačítko je vytvořena funkce *jmenoTlacitka_Click()*. Do této funkce je poté zapisován kód, který má probíhat po kliknutí na dané tlačítko.

V C# je v projektu pro přehlednost vytvořeno několik tříd. Rozdělení tříd u RSA šifry v této bakalářské práci odpovídá obrázku Obrázek 47.



Obrázek 47. Struktura projektu RSA šifry (C#)

Jelikož zde dochází ke generování velkého prvočísla, není dostačující zvolit datový typ int nebo long. Z toho vyplývá poměrně velká nevýhoda oproti Pythonu – musí se zde udávat datové typy. Je nutné nalézt vhodnou knihovnu nebo balíček, se kterým je možné generovat i velká prvočísla. Pro tuto potřebu je zde zvolen NuGet baliček (musí být do Visual Studia stažen přes možnost Spravovat balíčky NuGet u projektu) BouncyCastle.

Při šifrování je asi nejnáročnější generovat náhodná prvočísla. Balíček BouncyCastle nabízí datový typ BigInteger, který obsahuje metodu *ProbablePrime()*. Tato metoda však negeneruje prvočíslo se stoprocentní jistotou. Musí tedy docházet k ověření prvočísla. Níže (Obrázek 48) jsou uvedeny příklady, jak lze prvočíslo ověřit.

```
19          //dochazi ke generovani pravdepodobne random prvocisla p a q, jehoz prvociselnost se ovsem musi jeste overit
20          BigInteger p = BigInteger.ProbablePrime(48, random);
21          //dokud vygenerovane cislo neni prvocislem, generuje se nove
22          while (!jePrvocislo(p))
23          {
24              p = BigInteger.ProbablePrime(48, new Random());
25          }
26
27          //optimalizace overovani prvocisla
28          /* while (!RabinMiller.MillerRabin(p, 50))
29          {
30              p = BigInteger.ProbablePrime(48, random);
31          }*/
32
33          //misto funkce while vyse je vyuzito funkce pro overeni prvocisla - rabinMillerTest zabudovane jako metoda BigIntegeru
34
35          /*while (!p.RabinMillerTest(1, new Random()))
36          {
37              p = BigInteger.ProbablePrime(48, new Random());
38          }*/
39
```

Obrázek 48. Ověřování prvočísla (C#)

Řádek 20: generování náhodného pravděpodobně prvočísla

Řádek 22-25: ve while cyklu se generuje náhodné prvočíslo, dokud opravdu bude prvočíslem

Řádek 22: pro ověření prvočísla je využita metoda *jePrvocislo*(), která je uvedena na Obrázek 49. Tato metoda je z důvodu procházení od čísla 3 po malých skocích až po hodnotu velkého prvočísla velmi pomalá.

Řádek 28: další možnost ověření prvočísla – ve vytvořené třídě *RabinMiller.cs* je implementována metoda *MillerRabin*(), což už algoritmus viditelně urychlí

Řádek 35: zde je využita přímo metoda BigIntegeru

```
99      private bool jePrvocislo(BigInteger cislo)
100     {
101
102         if (!cislo.IsProbablePrime(1))
103         {
104             return false;
105         }
106         //sude cisla - nejsou prvocisla
107         BigInteger cislo2 = new BigInteger("2");
108         if (!cislo2.Equals(cislo) && BigInteger.Zero.Equals(cislo.Mod(cislo2)))
109         {
110             return false;
111         }
112
113          for (BigInteger i = new BigInteger("3"); i.Multiply(i).CompareTo(cislo) < 1; i = i.Add(cislo2))
114          {
115              if (BigInteger.Zero.Equals(cislo.Mod(i)))
116              {
117                  return false;
118              }
119          }
120         return true;
121     }
```

Obrázek 49. Funkce na ověření prvočísla (C#)

Pokud se využívá právě tohoto datového typu BigInteger, je zde rozdíl i v základních matematických operacích. Nelze použít přímo zápis krát (*), mínus (-) apod. BigInteger má zabudované metody jako *Multiply()* nebo *Substract()*. Jinak následné generování klíče *n* probíhá v principu stejně jako v Pythonu.

Rozdílem a opět více náročné oproti Pythonu je získávání klíče *e*. Musí být vygenerováno náhodné číslo BigInteger ve velkém rozsahu čísel. Nelze použít jednoduché metody jako *Random.Next()* – ty jsou vhodné pouze za použití datového typu int, což v tomto případě není dostačující. Proto je vytvořena další třída *RandomBigInteger.cs*, ve které je implementace generování náhodného čísla v daném rozsahu. Pro vyhodnocení toho, zda jsou čísla soudělná či nikoli, je využito metody BigIntegeru – *Gcd()* (řádek 76). Ukázka generování klíče *e* viz Obrázek 50.

```
67          BigInteger e = new BigInteger("1");
68          //trida slouzici ke generovani random BigIntegeru v urcitem rozsahu
69          RandomBigInteger randomCislo = new RandomBigInteger();
70          bool spatneE = true;
71          //dokud se nevygeneruje random cislo e, jehoz gcd(e,Fi) je rovno 1, generuje se dalsi
72          while (spatneE)
73          {
74              //generovani nahodneho BigInteger cisla v rozmezi 1 az Fi-1
75              e = randomCislo.NextBigInteger(jedna, Fi);
76              var gcd = e.Gcd(Fi);
77              if (gcd.CompareTo(jedna) != 0)
78              {
79                  spatneE = true;
80              }
81              else
82              {
83                  //pokud je gcd rovno 1, vypise se hodnota klice e do textBoxu a smycka while se ukonci
84                  klicE.Text = e.ToString();
85                  spatneE = false;
86              }
87
88          }
```

Obrázek 50. Generování klíče e – RSA šifra (C#)

Naopak výhodou oproti Pythonu je výpočet klíče *d*. Použitý datový typ BigInteger nabízí přímo funkci pro jeho výpočet (viz řádek 92 Obrázek 51). Vypisování textu uživateli probíhá pomocí zápisu *nazevPoleGUIKamZapsat.Text = text na vypsání*, jak lze vidět na řádku 94 níže.

```
89          //inverzni modulo
90          BigInteger minJedna = new BigInteger("-1");
91          //d se ziska pomoci vzorce d=e^-1 mod Fi
92          var d = e.ModPow(minJedna, Fi);
93          //klid D je zobrazen do textBoxu uzivateli
94          klicD.Text = d.ToString();
```

Obrázek 51. Výpočet klíče d – RSA šifra (C#)

Šifrování opět začíná ověřením, zda si uživatel přeje zachovat klíče nebo je generovat. Kontrola CheckBoxu probíhá na řádku 31 u Obrázek 52.

```
30          //overeni zda uzivatel zvolil, ze chce zachovat stejne nebo vlastni klice
31          if (checkBox1.Checked)
32          {//overeni, ze jsou vyplneny texboxy pro klice N a E a obsahuji pouze ciselne znaky
33              if (klicE.Text == "" || klicN.Text == "" || !klicN.Text.All(char.IsDigit) || !klicE.Text.All(char.IsDigit))
34              {
35                  upozorneni.Text = "Zadej prosím validní klíče.";
36                  return;
37              }
38          }
39          else
40          {
41              GenerovaniKlice klice = new GenerovaniKlice();
42              //pokud uzivatel chce spolu se sifrovanim generovat nove klice, vola se funkce, ktera to provede
43              klice.VygenerovatKlice(klicE, klicN, klicD);
44          }
```

Obrázek 52. Ověření CheckBoxu (C#)

Dále probíhá šifrování podle principu z vývojového diagramu. Další nevýhodou oproti Pythonu se zde jeví, že C# zřejmě nenabízí funkci pro rozdělení textu do podřetězců podle délky jako *re.findall()* použita v Pythonu. Při šifrování byl tedy pro průchod již binárním textem zvolen for cyklus znázorněný na Obrázek 53 spolu s ukázkou části funkce šifrování. Pro

získání textu zadaného uživatelem do formuláře se užívá zápisu: *proměnná=nazevPrvku-GUIzeKterehoChciCist.Text.*

```
26          string sifrovanyText = "";
27          //optimalizace string --> StringBuilder
28          //StringBuilder sifrovanyText = new StringBuilder();
29          //sifrovy text v binarnich cislech je prochazen po 88 (delce) nul a 1
30          int delkaOtBinFor = otbin.Length;
31          for (int i = 0; i < delkaOtBinFor; i += 88)
32          {//jestlize je delka otevereneho textu prevedeneho na binarni cisla delsi nebo rovna 88, do charrarray se ulozi delka 88znaku
33              if (delka >= 88)
34              {
35                  chararray = otbin.ToCharArray(i, 88);
36                  delka -= 88;
37              }
38              else
39              { //do charrarray se vlozi pouze zbyla delka znaku (mensi jak 88)
40                  chararray = otbin.ToCharArray(i, delka);
41
42              }
43              //chararray je preveden na string
44              string binarni88Cislo = new string(chararray);
45              //binarni cislo je prevedeno na BigInteger (cislo v desitkove soustave)
46              BigInteger cislo88Znaku = new BigInteger(binarni88Cislo, 2);
47
48              //je vypocitan sifrovy znak odpovidajici 8 znakum OT
49              BigInteger sifrZnak = cislo88Znaku.ModPow(e, n);
50
51              //do sifrovantText je pridan sifrovy znak
52              sifrovanyText += sifrZnak.ToString() + "\n";
53              //optimalizace do StringBuilderu je pridan sifrovy znak
54              //sifrovanyText.Append(sifrZnak.ToString() + "\n");
55
56          }
57          //po projiti celeho otevreneho textu prevedeneho na velke binarni cislo, je zasifrovany text zobrazen do textBoxu ve Formu
58          textZasifrovany.Text = sifrovanyText;
```

Obrázek 53. Ukázka části funkce šifrování RSA (C#)

Při dešifrování nejprve dochází k ověření klíče a textu (viz Obrázek 54).

```
66          //dojde k overeni, zda jsou vyplneny textboxy s klici slouzici k desifrovani a desifrovany text
67          if (klicDDesifr.Text == "" || klicNDesifr.Text == "" || !klicDDesifr.Text.All(char.IsDigit) || !klicNDesifr.Text.All(char.IsDigit) || textKDesifrovani.Text == "")
68          {
69              upozorneni.Text = "Zadejte validni klíče a text pro dešifrovaní. (pouze čísla)";
70              return;
71          }
```

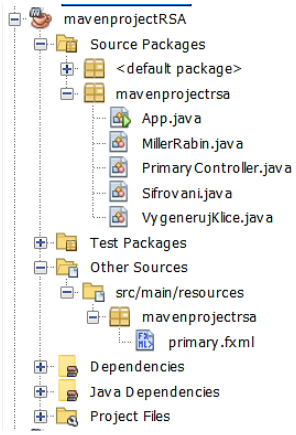
Obrázek 54. Ověření klíče a textu u dešifrování RSA (C#)

Další kód spolu s popisky pomocí komentářů je uveden v příloze P I.

C# má pro začátek výhodu, že postačí pro celý proces tvorby pouze jedno vývojové prostředí – Visual Studio. Všechno tím pádem lze upravovat v rámci jednoho projektu. Jako poměrně velká nevýhoda se jeví, že je potřeba dávat pozor na datové typy proměnných. Je zde složitější syntaxe (dodržování středníků, závorek). C# ve většině případů nenabízí tolik zabudovaných metod jako Python, tudíž musí docházet více k vypisování kódu, což nemusí být vždy přehledné. Konkrétně při implementaci šifry RSA zde byl z počátku problém zejména najít možnost generování velkého prvočísla, následné jeho ověření a dále generování náhodného čísla BigInteger v určitém rozsahu hodnot. V tomto případě to má Python vyřešeno lépe.

7.1.3 Java

Po vytvoření projektu ve vývojovém prostředí NetBeans (viz kapitola 6.3) je kód zapisován do souboru PrimaryController.java. Struktura projektu je znázorněna na Obrázek 55.



Obrázek 55. Struktura projektu RSA šifry (Java)

PrimaryController obsahuje definice proměnných, které se používají jako id u prvků GUI. Dále obsahuje funkce, které se taktéž mohou s GUI propojit. Ukázka třídy kontroleru je na Obrázek 56.

```
1  package mavenprojectrsa;
2
3  import java.io.IOException;
4  import javafx.fxml.FXML;
5  import javafx.scene.control.Button;
6  import javafx.scene.control.CheckBox;
7  import javafx.scene.control.Label;
8  import javafx.scene.control.TextArea;
9
10
11 public class PrimaryController {
12
13     @FXML
14     private TextArea klicE;
15     @FXML
16     private TextArea klicD;
17     @FXML
18     private TextArea klicN;
19     @FXML
20     private TextArea zasifrovanyText;
21     @FXML
22     private TextArea textKZasifrovani;
```

Obrázek 56. Ukázka třídy kontroleru (Java)

Řádek 3-8: potřebné importované knihovny

Řádek 13: klíčové slovo, které se užívá vždy před proměnnou nebo funkcí, která má být volána z GUI

Šifrování a dešifrování je velmi podobné s C#. Zde je výhoda, že datový typ BigInteger je použit z knihovny *java.math.BigInteger* a nemusí být složitě vyhledáván balíček jako v C#. Text zadaný uživatelem se z formuláře získá zápisem: *proměnná = nazevPrvkuZGUI.getText()*.

Při šifrování je opět nejprve ověřován CheckBox. K jeho hodnotě se lze dostat zápisem: *nazevCheckboxu.isSelected()*.

V Javě se pro průchod textem po úsecích podle délky nabízí využití *Splitteru* ze staženého balíčku guava a jeho importování: *import static com.google.common.base.Splitter.**

Průchod textem k šifrování již v binární podobě je vidět na Obrázek 57. Zobrazení textu uživateli probíhá zápisem podle řádku 48.

```
37          for (String binCislo88 :  fixedLength(88).split(binKSifrovani)){
38              //binarniCislo88 prevedeno na BigInteger
39              BigInteger bigIntCislo88 = new BigInteger(binCislo88, 2);
40              //sifrovany znak se dostane pomoci vzorce text^e mod n
41              BigInteger sifrZnak = bigIntCislo88.modPow(e, n);
42              //sifrovanyZnak je vlozen do kompletu sifrovanehoTextu
43              sifrovanyText+=sifrZnak+"\n";
44              //optimalizace String --> Stringbuiler
45              //sifrovanyText.append(sifrZnak).append("\n");
46          }
47          //zasifrovany text je zobrazen uzivateli
48          zasifrovanyText.setText(sifrovanyText);
```

Obrázek 57. Průchod textem v binární podobě a jeho šifrování

Při generování klíčů je použita knihovna *java.math.BigInteger* a *java.util.Random*. Datový typ BigInteger podobně jako v C# nabízí metodu, která pravděpodobně vygeneruje prvočíslo. To musí být opět ověřováno. Příklady ověření jsou znázorněny na Obrázek 58.

```
20          //vygeneruje se nejspise prvocislo - musi se pro jistotu dale overit
21          BigInteger p = BigInteger.probablePrime(48,new Random());
22
23          //dokud cisla p a q nebudou prvocisla, generuji se znova
24          while(!jePrvocislo(p)){
25              p= BigInteger.probablePrime(48,new Random());
26          }
27          //optimalizace overovani prvocisla pomoci rabinMiller
28          /*while(!MillerRabin.MillerRabin(p, 50)){
29              p= BigInteger.probablePrime(48,new Random());
30          }*/
```

Obrázek 58. Ověřování prvočísla – RSA (Java)

Řádek 21: funkce ověřování, zda se opravdu jedná o prvočíslo – podobná implementace jako v C#

Řádek 28: ověření prvočísla pomocí metody implementované v souboru MillerRabin.java

Zde je jedna malá nevýhoda oproti C#, datový typ nenabízí přímo funkci *RabinMillerTest()*.

Jinak je implementace podobná jako u jazyku C#. BigInteger obsahuje metody pro výpočet gcd, používá se místo klasického krát nebo mínus metod *multiply()* a *substract()* a klíč *d* je vypočten pomocí metody*: BigInteger d=e.modInverse(Fi)*.

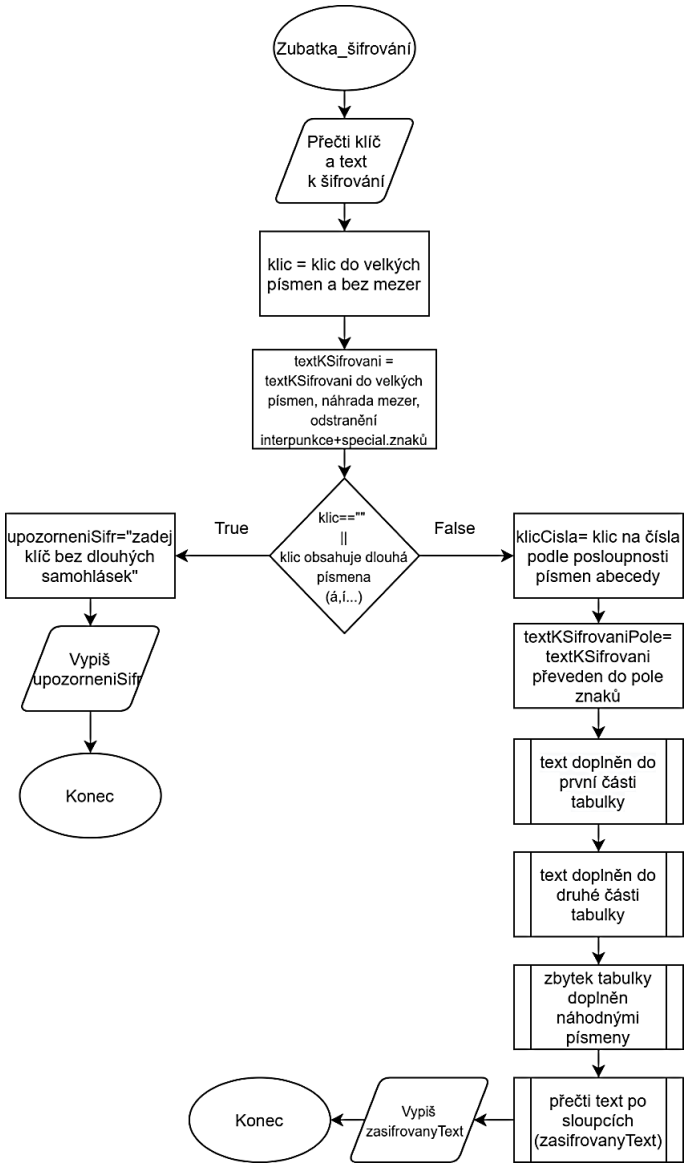
Java se dá v tomto případě téměř přirovnat k C#. Výhodou je, že se knihovna pro datový typ BigInteger, která využívá vhodných funkcí nemusela složitě hledat jako balíček u C#. Další

výhodou byla možnost využití balíčku guava pro rozdělení textu po úsecích dané délky. Nevýhodou je pouze možnost debugování, která v prostředí není na první pohled moc zřetelná.

7.2 Zubatka

Šifra Zubatka zde tvoří klasického zástupce transpoziční šifry. Jedná se o transpozici v tabulce a její teoretický princip je popsán v kapitole 3.2.5.

Struktura programu je vyjádřena prostřednictvím vývojových diagramů, které znázorňují přehledný postup šifrování a dešifrování. Podprogramy použité v diagramech jsou doloženy v příloze P I.



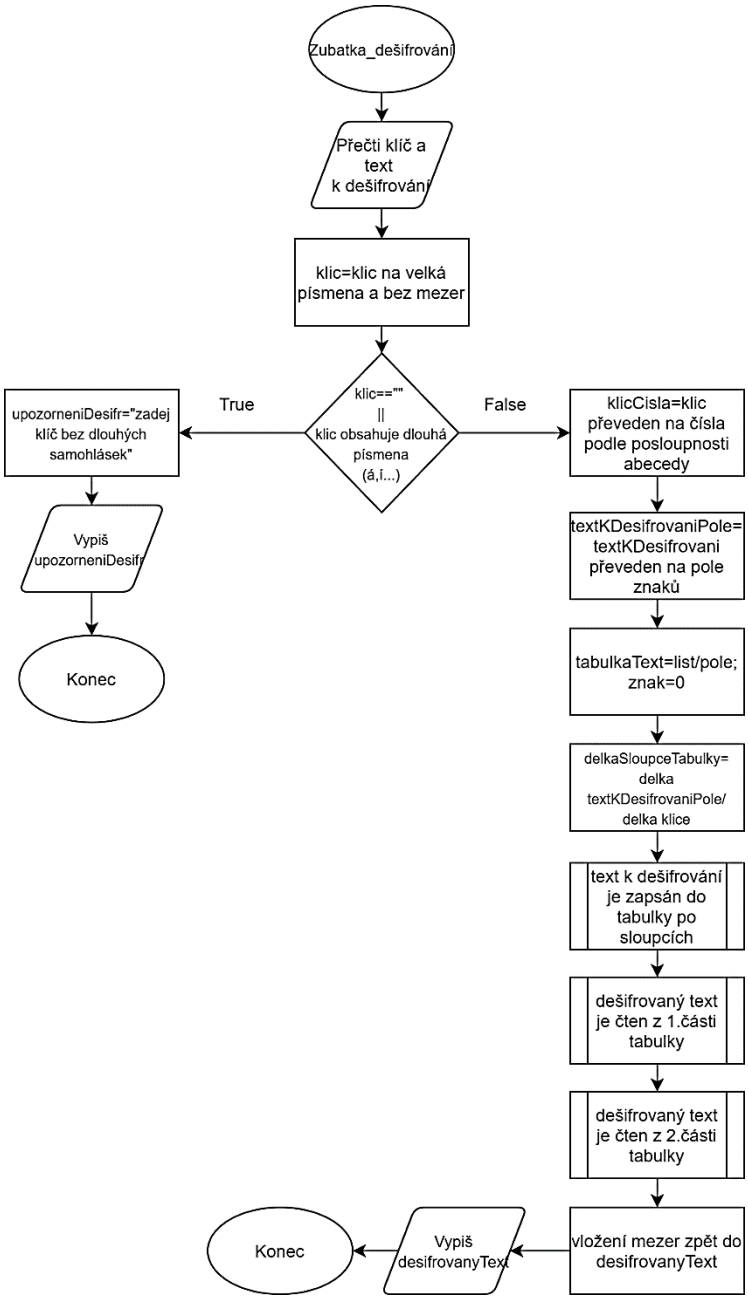
Obrázek 59. Vývojový diagram šifrování Zubatka

Při šifrování (viz Obrázek 59) je nejprve přečten klíč a text k šifrování ze vstupu uživatele. Klíč je následně převeden do velkých písmen a jsou v něm odstraněny případné mezery. Text k šifrování je taktéž převeden na velká písmena, jsou v něm nahrazeny mezery (v tomto případě pomocí řetězce „XMEX“). Dále je z textu odstraněna interpunkce, speciální znaky a čísla. Následně je ověřováno, zda zadaný klíč obsahuje dlouhá písmena. Pokud ano, uživatel je upozorněn a šifrování ukončeno. Pokud je klíč v pořádku, může dojít k samotnému šifrování. Klíč je převeden na čísla podle posloupnosti písmen v abecedě. Jestliže se v klíči nachází jedno písmeno vícekrát, odpovídá dvěma rozdílným po sobě jdoucím číslům. Dále je text k šifrování převeden do pole znaků. Následně dochází k naplnění první části tabulky[3] (kolik bude mít každý sloupec tabulky řádků je určeno klíčem převedeným na čísla viz Tabulka 19). Tabulka je procházena po řádcích. Než je do tabulky zapsán znak z pole znaků k zašifrování, je ověřováno, zda pozice v tabulce u konkrétního sloupce je menší než odpovídající hodnota v číselném klíči a je k dispozici znak, který má být do tabulky zapsán. Pokud znak již k dispozici není (délka textu k zašifrování je menší než hodnota indexu znak) a pořád se jedná o první část tabulky, je do ní zapsáno náhodné písmeno. Naplnění druhé části tabulky probíhá podobně. Opět se postupuje po řádcích, ale tentokrát dochází k ověřování, zda je hodnota počtu řádků konkrétního sloupce větší nebo rovna hodnotě odpovídající indexu sloupce v číselném klíči. Po doplnění tabulky textem k dešifrování je zbytek tabulky doplněn náhodnými písmeny tak, aby počet řádků ve všech sloupcích tabulky byl totožný. Je určena hodnota délky nejdelšího sloupce z tabulky. Poté je procházen každý sloupec v tabulce a doplňován náhodnými písmeny, dokud se jeho délka nerovná délce nejdelšího sloupce. Nakonec je celá tabulka procházena po sloupcích a jednotlivé znaky ukládány jako zašifrovaný text. Ten je poté vypsán uživateli a šifrování se ukončuje.

Dešifrování, jak lze vidět na diagramu na Obrázek 60, začíná načtením klíče a textu k dešifrování. Klíč je podobně jako u šifrování převeden na velká písmena bez mezer a ověřován, zda je prázdný nebo obsahuje dlouhá písmena. Pokud tomu tak je, uživatel je upozorněn a dešifrování se ukončuje. Pokud je klíč zadán v pořádku, je převeden na čísla podle posloupnosti v abecedě a text k dešifrování je převeden na pole znaků jako ve funkci šifrování. Počet sloupců tabulky je určeno délkou klíče a počet řádků určuje podíl délky textu a klíče. Text k dešifrování je tedy zapsán do tabulky po sloupcích klasicky přes cykly for. Následně

[3] Pro tabulku u šifry Zubatka je využito dvojrozměrného pole či listu listů

dochází k procházení tabulky po jednotlivý řádcích a pokud platí podmínka, že daný počet řádků ve sloupci (dané procházené indexy tabulky) je menší než odpovídající index v číselném klíči, uloží se znak k dešifrovanému textu. Dále je procházena tabulka opět po řádcích. Tentokrát aby se znaky ukládaly k dešifrovanému textu, musí být splněna podmínka, že počet řádků ve sloupci je větší nebo roven indexu v číselném klíči (přečtena druhá část tabulky). Na závěr jsou do textu vloženy zpět mezery – řetězec „XMEX“ je nahrazen mezerou a dešifrovaný text je vypsán uživateli do určeného pole ve formuláři.



Obrázek 60. Vývojový diagram dešifrování Zubatka

Jelikož je implementace této šifry v jednotlivých jazycích velmi podobná, jsou v následujících kapitolách popsány pouze některé rozdíly.

7.2.1 Python

Funkce nahrazení a normalizace textu použita při šifrování je znázorněna na Obrázek 61.

```
114     def NahrazeniANormalizaceTextu(self,nahrazeniMezery):
115         start=time()
116         #nahradit mezery a jine znaky
117         textNaZasifrovani=self.textKZasifrovani.toPlainText().upper().replace(" ",nahrazeniMezery)
118         #textNaZasifrovani=textNaZasifrovani.replace("1","JEDNA").replace("2","DVA").replace("3","TRI")
119         #normalizace textu - bez hacku, carek, znaku...
120         textNaZasifrovani=regex.sub(r'\p{Mn}', '', unicodedata.normalize('NFKD', textNaZasifrovani))
121 
122         textNaZasifrovani=re.sub('[^A-Z]+', '', textNaZasifrovani)+nahrazeniMezery
123         konec=time()-start
124         print("cas Zubatka nahrazaniANormalizaceTextu: "+str(konec))
125         return textNaZasifrovani
126 
```

Obrázek 61. Funkce nahrazení a normalizace textu (Python)

Řádky 115,123,124: slouží pouze k měření času – pro samotné šifrování nejsou potřebné

Řádek 117: text k zašifrování je převeden na velká písmena a jsou v něm nahrazeny mezery

Řádek 120: text je převeden do znaků bez interpunkce

Řádek 122: text je zbaven všech znaků kromě písmen A-Z a na jeho konec je doplněn řetězec znamenající mezeru

Další rozdíl a výhoda v Pythonu je při převodu klíče na čísla podle posloupnosti abecedy (viz Obrázek 62). Kromě přehlednějšího kódu je zde využito také metody *sorted()*, která přímo vrací znaky klíče setříděné dle abecedy. Taky je zde vidět jednodušší zápis použití listu, který má v sobě dostupné metody jako *count()* nebo *append()*.

```
128     def KlicNaCislaPodleAbecedy(self,klic):
129         start=time()
130         #setridene znaky klice podle abecedy
131         klicPodleAbecedy=sorted(klic)
132         #pro ulozeni cisel posloupnosti znaku klice
133         klicCisla = []
134         #pomocne pole - abych vedela kolik stejnych znaku je jiz ulozeno v klicCisla
135         klicZnakyPomocnePole=[]
136         #projdou se znaky puvodniho klice a na zaklade setrizenych znaku klice podle abecedy
137         #se jim prideli cislo podle posloupnosti do klicCisla
138         for i in klic:
139             cisloZnaku=klicPodleAbecedy.index(i)+1
140             if cisloZnaku in klicCisla:
141                 pocetZnaku=klicZnakyPomocnePole.count(i)
142                 klicCisla.append(cisloZnaku+pocetZnaku)
143             else:
144                 klicCisla.append(cisloZnaku)
145 
146             klicZnakyPomocnePole.append(i)
147 
148         konec=time()-start
149         print("cas Zubatka KlicNaCislaPodleAbecedy: "+str(konec))
150         return klicCisla
```

Obrázek 62. Funkce převodu klíčů na čísla podle posloupnosti abecedy (Python)

Dále je znázorněna část šifrování (Obrázek 63). Zde opět stačí pouze definování tabulky jednoduchým zápisem (řádek 42). V Pythonu je přehlednější zápis for cyklu a je umožněno jednoduchého zápisu procházení indexy v určitém rozmezí (*range()*).

Řádek 48, 49: pokud je při procházení tabulky nulový index i (nachází se na prvním řádku a žádné sloupce do tabulky ještě vloženy nebyly), dojde k vložení sloupce jako dalšího pole

Řádek 50: je ověřováno, zda je index řádku menší než hodnota v *klicCisla* na indexu jako index sloupce a stále je k dispozici text k doplnění do tabulky

Řádek 54: doplnění random písmene do první části tabulky, pokud již nejsou k dispozici znaky textu určeny k dešifrování – v Pythonu je potřeba pouze importování knihovny *random* a *string*, nemusí například docházet k ukládání proměnné s abecedou jako v Javě

```
27      def Zasifruj(self):
28          start = time()
29          self.upozorneniSifr.setText("")
30          self.textZasifrovany.setText("")
31          textNaZasifrovani=self.NahrazeniANormalizaceTextu("XMEX")
32          textNaZasifrovaniPole=self.StringNaList(textNaZasifrovani)
33          #optimalizace radek vyse pomoci funkce list
34          #textNaZasifrovaniPole=list(textNaZasifrovani)
35
36          klic=self.klicSifr.toPlainText().upper().replace(" ","")
37          if klic.isalpha() is False or klic=="" :
38              self.upozorneniSifr.setText("Zadejte klíč, který bude obsahovat pouze písmena abecedy.")
39              return
40
41          klicCisla=self.KlicNaCislaPodleAbecedy(klic)
42          tabulkaText=[]
43          self.znak=0
44          start1cast=time_ns()
45          #doplneni do prvni casti tabulky po radcich
46          for i in range(max(klicCisla)):
47              for j in range(len(klicCisla)):
48                  if i==0:
49                      tabulkaText.append([])
50                  if klicCisla[j] > i and self.znak <= len(textNaZasifrovaniPole)-1:
51                      tabulkaText[j].append(textNaZasifrovaniPole[self.znak])
52                      self.znak+=1
53                  elif i<klicCisla[j]:
54                      tabulkaText[j].append(random.choice(string.ascii_letters).upper())
55
```

Obrázek 63. Část funkce šifrování – Zubatka (Python)

V Pythonu je zde mimo jiné výhoda při optimalizaci for cyklů (viz kapitola 8.1).

Dalším rozdílem je například zjištění délky největšího sloupce v tabulce. V Pythonu stačí použít pouze funkci *max()* a *len()* pro nalezení nejdelšího sloupce a následné zjištění jeho délky (viz řádek 79 na Obrázek 64).

```
79          maximalniDelkaSloupce=len(max(tabulkaText, key=len))
80          for sloupecc in range(len(klic)):
81              while len(tabulkaText[sloupecc])<maximalniDelkaSloupce:
82                  tabulkaText[sloupecc].append(random.choice(string.ascii_letters).upper())
```

Obrázek 64. Zjištění velikosti nejdelšího sloupce v tabulce (Python)

7.2.2 C#

V C# je funkce nahrazení a normalizace textu, která je využita při šifrování, zapsána poněkud složitěji (oproti Pythonu). Jak lze vidět na řádcích 97-99 na Obrázek 65, je zapotřebí zejména složitějšího zápisu při odstraňování interpunkce z textu. Použití třídy *Stopwatch* slouží pro měření doby trvání funkce. Pro účel šifrování není potřeba.

```
89      private String NahrazeniANormalizaceTextu()
90      {
91          Stopwatch stopwatch = Stopwatch.StartNew();
92          //nahradit mezery a jine znaky
93          string textNaZasifrovani = textKZasifrovani.Text.ToUpper().Replace(" ", "XMEX");
94          //textNaZasifrovani=textNaZasifrovani.replace("1","JEDNA").replace("2","DVA").replace("3","TRI")..
95
96          //normalizace textu - bez hacku, carek,
97          textNaZasifrovani = textNaZasifrovani.Normalize(NormalizationForm.FormD);
98          var chars = textNaZasifrovani.Where(c => CharUnicodeInfo.GetUnicodeCategory(c) != UnicodeCategory.NonSpacingMark).ToArray();
99          textNaZasifrovani = new string(chars).Normalize(NormalizationForm.FormC);
100
101         //zachovani pouze pismen (bez ,/ znaku...)
102         textNaZasifrovani = Regex.Replace(textNaZasifrovani,"[^A-Z]+", "") + "XMEX";
103
104         stopwatch.Stop();
105         Console.WriteLine("NahrazeniANormalizaceTextu cas: " + stopwatch.ElapsedTicks);
106         return textNaZasifrovani;
107
108     }
```

Obrázek 65. Funkce nahrazení a normalizace textu (C#)

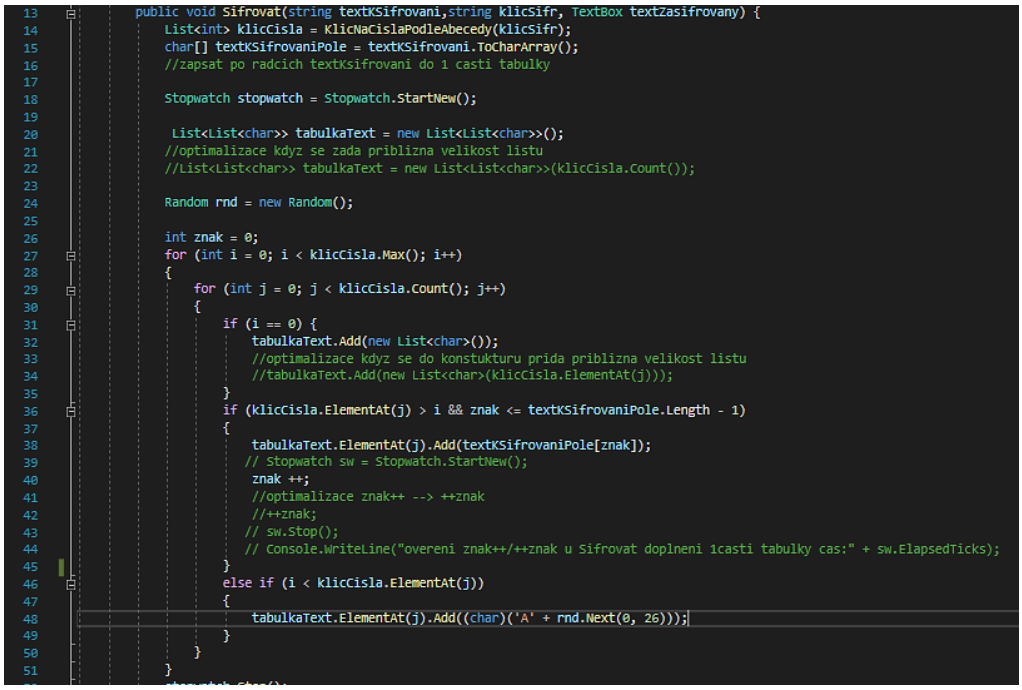
V C# ve funkci převedení klíče do číselné podoby na základě posloupnosti písmen v abecedě (viz Obrázek 66) je nevýhoda, že musí docházet k použití Listu. Klasické definování pole zde nestačí, jelikož na něj nelze použít funkce jako *Add()* nebo nalezení indexu podle daného znaku. Proces setřídění znaků klíče podle abecedy (řádek 116-117) je taktéž zapsán složitěji ve srovnání s Pythonem.

```
114     private List<int> KlicNaCislaPodleAbecedy(string klic) {
115         Stopwatch stopwacth = Stopwatch.StartNew();
116         char[] poleZnakuKlicePodleAbecedy = klic.ToCharArray();
117         Array.Sort(poleZnakuKlicePodleAbecedy);
118         List<int> klicCisla = new List<int>();
119         List<char> klicZnakyPomocne = new List<char>();
120         //optimalizace -  do konstrukturu listu se zada predem cca budouci velikost velikost
121         //List<int> klicCisla = new List<int>(klic.Length);
122         //List<char> klicZnakyPomocne = new List<char>(klic.Length);
123
124         Stopwatch sw = Stopwatch.StartNew();
125         foreach (char znak in klic.ToCharArray())
126         {
127             int cisloZnaku = poleZnakuKlicePodleAbecedy.ToList().IndexOf(znak)+1;
128             if (klicCisla.Contains(cisloZnaku))
129             {
130                 int pocetZnaku=klicZnakyPomocne.Count(znak.Equals);
131                 klicCisla.Add(cisloZnaku + pocetZnaku);
132             }
133             else
134             {
135                 klicCisla.Add(cisloZnaku);
136             }
137             klicZnakyPomocne.Add(znak);
138         }
```

Obrázek 66. Ukázka převodu klíče na čísla podle abecedy (C#)

U šifrování v C# musí při definování tabulky dojít k použití listu složeného z listů znaků. Stejně jako v Pythonu jsou pro průchod tabulkou použity for cykly, jejichž zápis se zde ale zdá méně přehledný. Jelikož došlo k použití listu, je i zde možnost přidávání prvků do

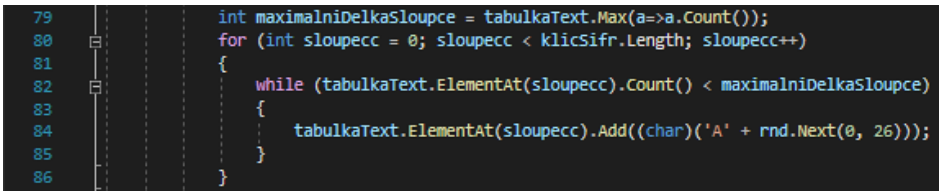
sloupců, i když nebyla dříve definována jejich délka. Pro generování random písmene je zde použita pouze třída *Random* (řádek 24, 48 na Obrázek 67).



```
13      public void Sifrovat(string textKSifrovani,string klicSifr, TextBox textZasifrovany) {
14          List<int> klicCisla = KlicNaCislaPodleAbecedy(klicSifr);
15          char[] textKSifrovaniPole = textKSifrovani.ToCharArray();
16          //zapsat po radcich textKsifrovani do 1 casti tabulky
17
18          Stopwatch stopwatch = Stopwatch.StartNew();
19
20           List<List<char>> tabulkaText = new List<List<char>>();
21          //optimalizace kdyz se zada priblizna velikost listu
22          //List<List<char>> tabulkaText = new List<List<char>>(klicCisla.Count());
23
24          Random rnd = new Random();
25
26          int znak = 0;
27          for (int i = 0; i < klicCisla.Max(); i++)
28          {
29              for (int j = 0; j < klicCisla.Count(); j++)
30              {
31                  if (i == 0) {
32                      tabulkaText.Add(new List<char>());
33                      //optimalizace kdyz se do konstukturu prida priblizna velikost listu
34                      //tabulkaText.Add(new List<char>(klicCisla.ElementAt(j)));
35                  }
36                  if (klicCisla.ElementAt(j) > i && znak <= textKSifrovaniPole.Length - 1)
37                  {
38                      tabulkaText.ElementAt(j).Add(textKSifrovaniPole[znak]);
39                     // Stopwatch sw = Stopwatch.StartNew();
40                      znak ++;
41                      //optimalizace znak++ --> ++znak
42                      //++znak;
43                     // sw.Stop();
44                     // Console.WriteLine("overeni znak++/++znak u Sifrovat doplneni 1casti tabulky cas:" + sw.ElapsedTicks);
45                  }
46                  else if (i < klicCisla.ElementAt(j))
47                  {
48                      tabulkaText.ElementAt(j).Add((char)('A' + rnd.Next(0, 26)));
49                  }
50              }
51          }
```

Obrázek 67. Ukázka části šifrování – Zubatka (C#)

Zjištění délky největšího sloupce v tabulce zde probíhá pomocí funkce listu *Max()*, kde je v jeho argumentu lambda výraz (řádek 79 na Obrázek 68) – složitější na zápis než Python, jednodušší oproti Javě.



```
79                  int maximalniDelkaSloupce = tabulkaText.Max(a=>a.Count());
80                  for (int sloupecc = 0; sloupecc < klicSifr.Length; sloupecc++)
81                  {
82                      while (tabulkaText.ElementAt(sloupecc).Count() < maximalniDelkaSloupce)
83                      {
84                          tabulkaText.ElementAt(sloupecc).Add((char)('A' + rnd.Next(0, 26)));
85                      }
86                  }
```

Obrázek 68. Zjištění velikosti nejdelšího sloupce v tabulce (C#)

7.2.3 Java

V Javě se naopak funkce pro nahrazení a normalizaci textu podobá přehledné implementaci jako v Pythonu (viz Obrázek 69).

```
87      private String NahrazeniANormalizaceTextu(){
88          long start = System.nanoTime();
89           //nahradit mezery a jine znaky
90          String textNaZasifrovani=textKSifrovani.getText().toUpperCase().replace(" ", "XMEX");
91          //textNaZasifrovani=textNaZasifrovani.replace("1","JEDNA").replace("2","DVA").replace("3","TRI")..
92
93          //normalizace textu - bez hacku, carek, znaku...
94          textNaZasifrovani=Normalizer.normalize(textNaZasifrovani, Normalizer.Form.NFD);
95         //zachovani pouze pismen
96          textNaZasifrovani=textNaZasifrovani.replaceAll("[^A-Z]+", "")+"XMEX";
97          long konec=System.nanoTime()-start;
98          System.out.print("Zubatka NahrazeniANormalizaceTextu cas: "+konec+"\n");
99
100         return textNaZasifrovani;
101     }
```

Obrázek 69. Funkce nahrazení a normalizace textu (Java)

V Javě je funkce převodu klíče na čísla podle posloupnosti abecedy srovnatelná s funkcí v C#. Funkce šifrování probíhá taktéž podobným způsobem jako v C#. Jako tabulky je využito listu listů znaků. Pro určení maximální hodnoty z listu *klicCisla* (řádek 36 Obrázek 70) je importována knihovna *java.util.Collections*. Mírnou nevýhodou je zde generování náhodného písmene. Je vytvořena proměnná, která v sobě obsahuje znaky abecedy (řádek 26) a podobně jako v C# je použita třída *Random* (řádek 27 a 50).

```
21  public class Sifrovani {
22      public void Zasifruj(String textKSifrovani, String klicSifr, TextArea textZasifrovany ){
23          char[]textKSifrovaniPole=textKSifrovani.toCharArray();
24          List<Integer> klicCisla=KlicNaCislaPodleAbecedy(klicSifr);
25           //na generovani random pismene do tabulky
26          String abeceda = "ABCDEFGHIJKLMNOPQRSTUVWXYZ";
27          Random rd = new Random();
28          //doplneni prvni casti tabulky po radcich
29          long start=System.nanoTime();
30          //List<List>tabulkaText=new ArrayList<>();
31          //optimalizace - stanoveni priblizne velikosti listu
32          List<List>tabulkaText=new ArrayList<>(klicCisla.size());
33
34          int znak=0;
35
36          for (int i = 0; i < Collections.max(klicCisla,null); i++){
37              for (int j = 0; j < klicCisla.size(); j++){
38                  if (i == 0) {
39                      //tabulkaText.add(new ArrayList<Character>());
40                      //optimalizace - stanoveni priblizne velikosti listu
41                      tabulkaText.add(new ArrayList<Character>(klicCisla.get(j)));
42                  }
43                  if (klicCisla.get(j) > i && znak <= textKSifrovaniPole.length - 1)
44                  {
45                      tabulkaText.get(j).add(textKSifrovaniPole[znak]);
46                      znak ++;
47                  }
48                  else if (i < klicCisla.get(j))
49                  {
50                      tabulkaText.get(j).add(abeceda.charAt(rd.nextInt(abeceda.length())));
51                  }
52              }
53          }
```

Obrázek 70. Ukázka části šifrování – Zubatka (Java)

V Javě probíhá zjištění maximální délky sloupce tabulky poněkud složitěji, a to pomocí knihovny *java.util.Optional* (viz Obrázek 71).

```
109             Optional<List> maximalniDelkaSloupce = tabulkaText.stream()
110                 .sorted((e1, e2) -> e1.size() > e2.size() ? -1 : 1)
111                 .findFirst();
112
113             for(int sloupecc=0;sloupecc<klicSifr.length();sloupecc++){
114                 while(tabulkaText.get(sloupecc).size()<maximalniDelkaSloupce.get().size()){
115                     tabulkaText.get(sloupecc).add(abeceda.charAt(rd.nextInt(abeceda.length())));
116                 }
117             }
```

Obrázek 71. Zjištění velikosti nejdelšího sloupce v tabulce (Java)

Řádek 109-111: vrací se nejdelší sloupec

Řádek 114: *maximalniDelkaSloupce.get().size()* – je zjištěna velikost nejdelšího sloupce

7.3 One – time Pad

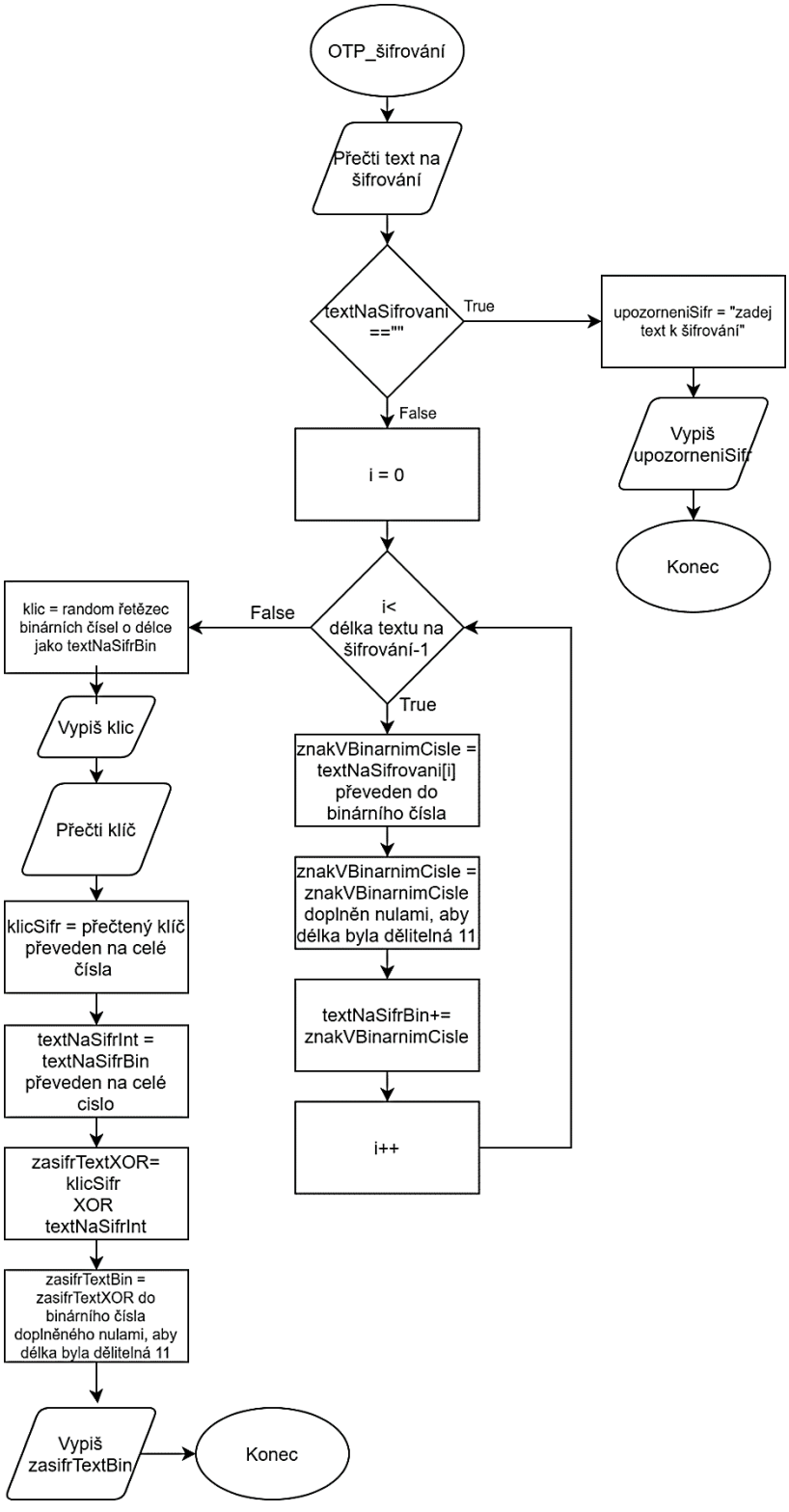
Šifra One – time Pad neboli Vernamova šifra tvoří zástupce symetrických proudových šifer. Zde je popsána varianta, při které dochází k převodu textu do binárních čísel, vygenerování náhodného binárního klíče stejně dlouhého jako text k zašifrování a následné operaci XOR mezi klíčem a původním textem. Zašifrovaný text je tvořen výsledkem provedené operace XOR. Text je poté dešifrován pomocí stejného klíče, jako byl použit na zašifrování. Operace XOR je tentokrát volána nad textem určeným k dešifrování v binární podobě a daným klíčem. Výsledkem je dešifrovaný text, který je dále převeden z binárního tvaru zpět na text.

Průběh šifrování a dešifrování je znázorněn pomocí diagramů níže (Obrázek 72 a Obrázek 73).

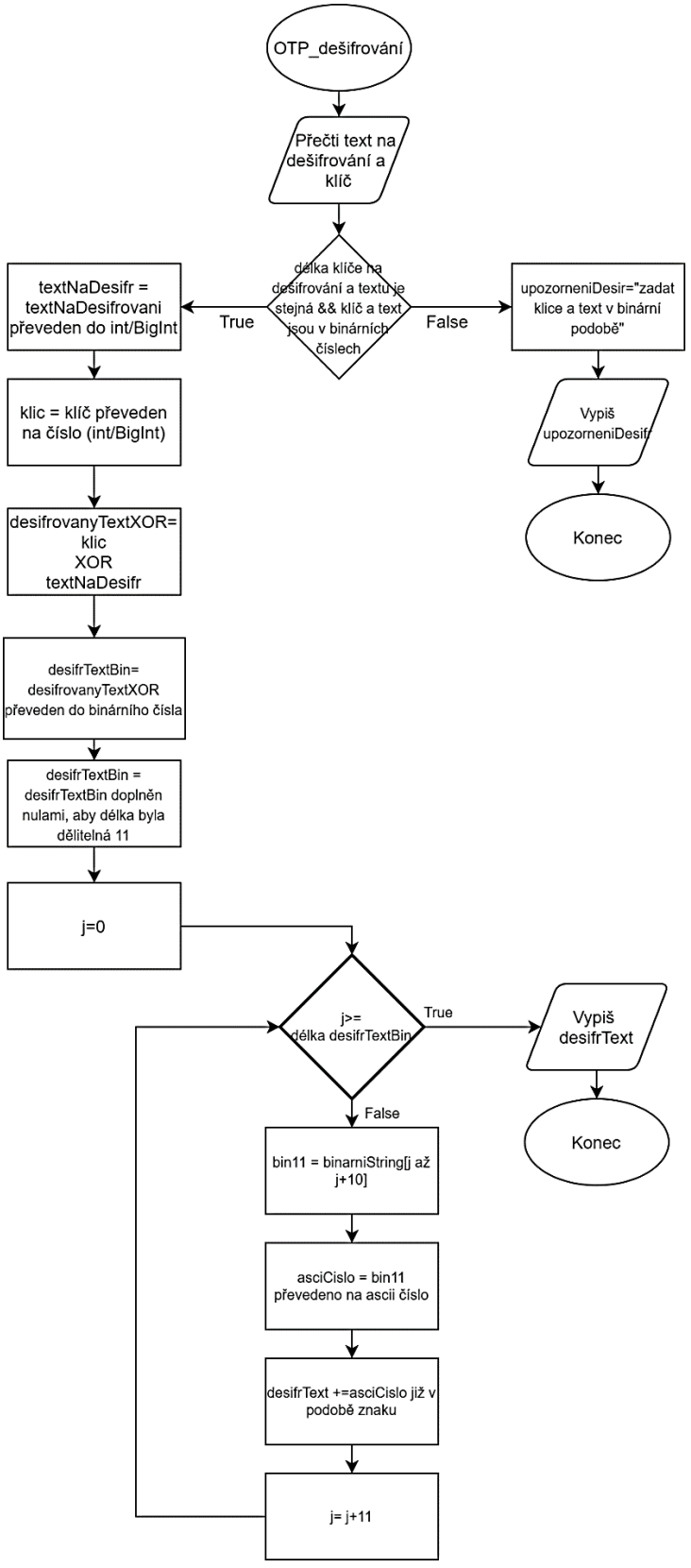
Šifrování začíná načtením textu určeného k zašifrování ze vstupu od uživatele. Pokud je text prázdný, uživateli je zobrazeno upozornění a funkce šifrování je ukončena. Jestliže je text zadán v pořádku je každý jeho znak převeden do binárního čísla, které je doplněno nulami na začátek tak, aby jeho délka byla dělitelná 11. Každý znak v binární podobě je ukládán do celkového binárního textu. Následně je vygenerován klíč – náhodné binární číslo, jehož délka odpovídá délce celkového binárního textu. Aby mohla být zavolána operace XOR, je klíč a text k zašifrování v binární podobě převeden do dekadického čísla (int nebo BigInteger). Zašifrovaný text je výsledkem operace XOR nad klíčem a textem k zašifrování již v podobě čísla dekadického. Zašifrovaný text je převeden do binárního čísla a doplněn nulami, aby jeho délka byla opět dělitelná 11. Takto zašifrovaný text je vypsán uživateli a funkce šifrování je ukončena.

Dešifrování je znázorněno diagramem na Obrázek 73. Nejprve je načten klíč a text k dešifrování. Následně dochází k ověření, zda jsou stejně dlouhé a v binární podobě. Pokud ne,

uživateli je vypsáno upozornění a dešifrování končí. Pokud jsou podmínky splněny, jsou klíč i text k dešifrování převedeny do decimálního čísla a je nad nimi zavolána operace XOR, což vrací již dešifrovaný text v podobě decimálního čísla. Toto číslo je převedeno na číslo binární a jsou mu doplněny nuly na začátek tak, aby jeho délka byla dělitelná 11. Dalším krokem je průchod výsledným binárním číslem po úsecích s délkou 11 a jejich převedení do decimálního čísla a následně do znaku, který je uložen ke kompletnímu dešifrovanému textu. Po převedení binárního čísla do textu je dešifrovaný text vypsán uživateli a funkce dešifrování končí.



Obrázek 72. Vývojový diagram šifrování OTP



Obrázek 73. Vývojový diagram dešifrování OTP

Dále jsou podobně jako u ostatních šifer uvedeny některé z rozdílů v implementaci v různých jazycích.

7.3.1 Python

Při šifrování je výhodou, že nemusí být zadávány datové typy. Jednoduše se pouze zapíše název proměnné. Další výhodou je přehledný a intuitivní zápis funkcí na převedení binárních čísel do decimálních a provedení operace XOR (řádek 68 Obrázek 74).

```
61          #text preveden na binarni cislo
62          textKSifrovaniBinarni = self.TextToBin(textKSifrovani)
63          #vygenerovan klic binarnich cisel o stejne delce jako text k zasifrovani
64          klic=self.VygenerujKlic(len(textKSifrovaniBinarni))
65          #klic je zobrazen uzivateli
66          self.klicSifr.setText(klic)
67          #operace XOR
68          zasifrBinText=int(textKSifrovaniBinarni,2) ^ int(klic,2)
```

Obrázek 74. Část funkce šifrování – OTP (Python)

U funkce, která zajišťuje vložení nul na začátek řetězce tak, aby jeho délka byla dělitelná 11, lze říct, že je o něco málo čitelnější než u C#. Jedná se tak o jednu z mála výjimek a je zobrazena na Obrázek 75.

```
36      #vklada nuly dokud delka stringu neni delitelna 11
37      def vlozNuly(self,string):
38          while len(string)%11 > 0:
39              string=string[:0] + "0" + string[0:]
40          return string
```

Obrázek 75. Funkce vkládání nul na začátek řetězce (Python)

U převodu binárních čísel s délkou 11 do textu znaků je zde nejprve využito funkce *re.findall()* pro rozdělení delšího textu v binární podobě do čísel s délkou 11. Následný převod každého takového binárního čísla do znaku je znázorněn na řádcích 122-125 na Obrázek 76. Jedná se o přehledný zápis a opět intuitivní využití funkcí. Další výhodou je také možnost optimalizace for cyklů zmíněna v kapitole 5.1.

```
120         rozdelenePo11=re.findall('.{1,11}',bintext)
121         text=''
122         for i in rozdelenePo11:
123                 asc=int(i, 2)
124                 znak=chr(asc)
125                 text+=znak
```

Obrázek 76. Převod binárního čísla do znaku

7.3.2 C#

V C# byl pro převedení binárních čísel do decimálních zvolen datový typ BigInteger. Tento datový typ podobně jako u RSA je z NuGet balíčku BouncyCastle (musí být použit *using Org.BouncyCastle.Math*). I samotný zápis procesu převedení binárního čísla do decimálního

je složitější a delší. Nejprve dochází k převodu na BigInteger (řádek 23 a 24 Obrázek 77) a až následně je zavolána operace XOR, která je zde použita jako metoda třídy datového typu (řádek 25).

```
19          string textNaSifrBin = StringDoBinarnichCisel(textNaSifrovani);
20          string klicSifr = VygenerujKlic(textNaSifrBin.Length);
21          klic.Text=klicSifr;
22
23          BigInteger klicSifrBigInt = new BigInteger(klicSifr, 2);
24          BigInteger textNaSifrBigInteger = new BigInteger(textNaSifrBin, 2);
25          BigInteger zasifrTextXOR = textNaSifrBigInteger.Xor(klicSifrBigInt);
```

Obrázek 77. Část šifrování – OTP (C#)

Jak již bylo zmíněno v oblasti jazyka Python, zápis funkce pro vkládání nul na začátek řetězce je zde snadněji pochopitelná (viz Obrázek 78). A to díky tomu, že datový typ string přímo nabízí funkci pro vložení požadovaného znaku na určitý index (funkce *Insert()*).

```
66      //doplneni nul na zacatek stringu tak, aby jeho delka byla delitelna 11
        Počet odkazů: 2
67      private string DoplnNulyDoBin(string s)
68      {
69          while (s.Length % 11 > 0)
70          {
71              s = s.Insert(0, "0");
72          }
73          return s;
74      }
```

Obrázek 78. Funkce vkládání nul na začátek řetězce (C#)

Jak lze vidět na Obrázek 79, funkce pro převod binárního řetězce do textu znaků je méně přehledná a náročnější na pochopení než v Pythonu. Jelikož C# nenabízí funkci jako *re.findall()* v Pythonu, je zde postupně procházen vstupní string, kterému je vždy odstraněno prvních 11 znaků (řádek 120). Těchto prvních 11 znaků je uloženo do proměnné *bin11* (řádek 118), následně převedeno na decimální číslo (řádek 122) a přidáno do vraceného řetězce v podobě znaku (řádek 126).

```
116         while (s.Length > 0)
117         {//rozdeleni cisla po 11
118             var bin11 = s.Substring(0, 11);
119             //do s se ulozi s bez 11 prvnich znaku, ktere se budou prevadet
120             s = s.Substring(11);
121             //binarni cislo je prevedeno na asciCislo
122             var asciCislo = Convert.ToInt32(bin11, 2);
123             //do sb je vlozen znak (char) odpovidajici asciCislu
124            // sb += (char)asciCislo;
125             //optimalizace string --> StringBuilder
126             sb.Append((char)asciCislo);
127         }
```

Obrázek 79. Převod binárního řetězce do textu znaků (C#)

7.3.3 Java

V Javě je zápis kódu pro převedení binárních čísel do decimálních téměř totožný se zápisem v C# (viz Obrázek 80). Hlavním rozdílem je, že pro použití datového typu BigInteger je zde

potřeba využít pouze knihovny *java.math.BigInteger* (nemusí docházet ke stahování balíčku jako u C#). Ve srovnání s Pythonem je zde zápis znovu o něco náročnější.

```
26          String textKSifrovaniBin = StringdoBinString(textKSifrovani);
27          String klic=vygenerujBinKlic(textKSifrovaniBin.length());
28          klicSifr.setText(klic);
29
30          BigInteger textKSifrovaniBigInt = new BigInteger(textKSifrovaniBin,2);
31          BigInteger klicBigInt=new BigInteger(klic,2);
32          BigInteger zasifrTextBigInt = textKSifrovaniBigInt.xor(klicBigInt);
```

Obrázek 80. Část funkce šifrování (Java)

V Javě je funkce, která obstarává doplňování nul podobná zápisu v C#. Rozdílem je, že navíc dochází k převodu Stringu do StringBuilderu (viz Obrázek 81). Tento krok je zaveden proto, aby mohla být použita funkce *insert()*. Nakonec nestačí vrátit pouze StringBuilder – musí být převeden zpět na String pomocí funkce *toString()*.

```
84      //doplni nuly do stringu binarnich cisel tak, aby jeho delka byla delitelna 11
85      private String DoplnNulyDoBin(String s){
86          StringBuilder sb = new StringBuilder(s);
87          while (sb.length() % 11 > 0) {
88              sb=sb.insert(0, "0");
89          }
90          return sb.toString();
91          }
```

Obrázek 81. Funkce vkládání nul na začátek řetězce (Java)

Funkce pro převod binárního řetězce do řetězce znaků (Obrázek 82) probíhá v Javě téměř totožně jako v C#. Liší se pouze v použití tříd, kdy Java využívá třídy Integer s funkcí *parseInt()* zatímco v C# je využita třída *System.Convert* s metodou *ToInt32()*. V tomto případě je opět nejpřehlednější a nejméně náročné z hlediska práce použití Pythonu.

```
112         while (b.length()> 0) {
113             //binarni cislo je po delce 11 prevedeno na pismena, ktera jsou vracena v podobe stringu
114                 var bin11 = b.substring(0, 11);
115                 b = b.substring(11);
116                 var asciCislo = Integer.parseInt(bin11, 2);
117                 sb+=(char)asciCislo;
118                 //optimalizace String --> StringBuilder
119                 //sb.append((char)asciCislo);
120             }
```

Obrázek 82. Převod binárního řetězce do textu znaků (Java)

7.4 Shrnutí výhod a nevýhod implementace šifer v daných jazycích

Z hlediska přehlednosti a čitelnosti kódu a počtu řádků je nejlepší volbou Python. Má velkou výhodu v tom, že nemusí docházet ke psaní středníků na konci řádku a ani například ke psaní složených závorek. Místo závorek je příslušnost částí kódu k daným oblastem (zda se jedná

o kód náležící například stále for cyklu, podmínce nebo se jedná již zase o základní tok programu) dána odsazením od začátku řádku. Tímto je i docíleno přehlednosti kódu, programátor musí dodržovat dané odsazování. Nevýhodou se může stát, že v případě špatného odsazení může dojít k tomu, že se kód nezkompiluje, a chyba se hledá obtížněji než chybějící složená závorka. Jeho syntaxe je jednodušší než syntaxe C# nebo Javy. Délka programu z hlediska počtů řádků se v Pythonu taktéž velmi redukuje. Zejména, když dochází k optimalizaci kódu pomocí List Comprehension nebo funkcí jako mapování či užití lambda výrazů. Zde lze například for cykly i s podmínkami místo rozepisování na několik řádků sepsat pouze do jednoho.

Přehlednost a čitelnost kódu zapsaném v C# nebo Javě je o něco menší než u Pythonu. Aby byl program přehlednější je dobré oddělovat funkce prázdnými řádky nebo potřebné závorky psát vždy na samostatný řádek a dodržovat odsazování, které je zde nepovinné, ale rozhodně se jím usnadní následné čtení kódu. Z hlediska počtu řádků programu nelze dosáhnou takové redukce jako v Pythonu. Každý krok programu je zapisován na nový řádek a většinou tyto jazyky nemají dostupných tolik knihoven a funkcí, takže se zápis kódu stává složitějším (místo použití jedné metody musí dojít k definování celé funkce ručně).

Jak již bylo zmíněno výše Python obsahuje velké množství knihoven, které velmi usnadňují práci. Při psaní šifer to lze vidět zejména u šifry RSA, kdy je generování prvočísel nebo random čísla z určitého intervalu díky využití knihovních funkcí poměrně snadné. Další výhodou je, že nemusí docházet k udávání datových typů.

V C# podobně jako v Javě datové typy proměnných musí být uvedeny, což může vést k nutnosti náročnějšího hledání vhodného datového typu, popřípadě knihovny či NuGet balíčků, který daný typ nabízí. Toto hledání bylo podstoupeno například u šifry RSA, kde byla zapotřebí práce s velkými čísly. Práce v C# je tedy časově náročnější. V možnosti použití datových typů je v tomto případě (při tvorbě daných šifer) jednodušší nalezení vhodného datového typu v Javě. V oblasti použití funkcí již nalezeného datového typu či nutnosti psaní vlastních funkcí, pro které nejsou vestavěné metody dostupné, jsou tyto dva jazyky na srovnatelné úrovni.

Následující odstavce se věnují porovnání použitých vývojových prostředí z hlediska snadnosti orientace, možnosti debugování či našeptávání kódu. Vývojové prostředí, které je vybráno pro jazyk Python (Spyder) působí minimalisticky a přehledně. Velmi snadno se s ním začíná pracovat. Podporuje barevné zvýrazňování syntaxe i našeptávání kódu, který je

možné dále použít (například při zápisu proměnných, které již jsou definovány nebo při použití funkcí). Dále je zde možnost ladění kódu, díky čemuž se snadněji hledají případné chyby, a průzkumník proměnných, ve kterém jsou uváděny všechny proměnné s informacemi o svém typu, velikosti a uložené hodnotě.

Vývojové prostředí Visual Studio využito pro práci s jazykem C# je taktéž velmi přehledné a práce zde je intuitivní. Kromě anglického jazyka nabízí možnost mít prostředí i v jazyce českém. Taktéž podporuje zvýrazňování syntaxe i našeptávání kódu, které se zdá být propracovanější než ve Spyderu. Podporuje i ladění kódu, které umožňuje nalezení chybně napsaného kódu či zjištění, co je postupně do proměnných ukládáno.

Apache NetBeans IDE, vývojové prostředí zvoleno pro práci s jazykem Java, je ve srovnání s IDE použitými u ostatních jazyků pro první použití méně přehledné, zejména z hlediska možnosti ladění kódu. Zvýrazňování syntaxe je zde opět podporováno. Doplňování kódu (našeptávání) konkrétně při používání proměnných nebo funkcí, které již byly definovány, zde bohužel není tak automatické jako v ostatních jazycích. Aby se zobrazily dané možnosti, musí dojít k zapsání dvou teček.

Souhrn výhod a nevýhod je taktéž zobrazen pomocí tabulky níže (Tabulka 24).

Tabulka 24. Souhrn výhod a nevýhod vybraných jazyků

	Python	**C#**	**Java**
Přehlednost	+ velmi přehledný a jednoduše čitelný kód + použitím optimalizace velká redukce řádků	+ přehledný – kód zapisován na větší množství řádků +– s použitím odsazování čitelný (pro začátek však méně jak Python)	+ přehledný – kód zapisován na větší množství řádků +– s použitím odsazování čitelný (pro začátek však méně jak Python)
Syntaxe	+ jednoduchá syntaxe	– syntaxe složitější než Python – nutnost udávání datových typů	– složitější syntaxe oproti Pythonu – nutnost udávání datových typů

	+ nepoužívání středníků nebo složených závorek – nutné si zvyknout na odsazování používané namísto závorek + nemusí se udávat datové typy	+– používání složených závorek – nutnost použití středníků	+– používání složených závorek – nutnost použití středníků
Knihovny, funkce…	+ velké možnosti použití knihoven + velmi intuitivní používání funkcí + velké množství funkcí pro práci s textem, které je při šifrování potřeba	– složitější nalezení vhodné knihovny nebo funkce, která je potřebná – není dostupné tolik knihovních funkcí jako v Pythonu, mnohdy je nutné je psát ručně – složité nalezení vhodného datového typu (RSA šifra)	– složitější nalezení vhodné knihovny nebo funkce, která je potřebná – není dostupné tolik knihovních funkcí jako v Pythonu, mnohdy je potřeba je psát ručně + poměrně snadné nalezení vhodného datového typu (RSA šifra)
Jak snadno se hledali zdroje (pomoc, návody…)	+ snadné hledání zdrojů jako informace k daným funkcím nebo návodům, jak lze určitý problém řešit. Většinou stačilo nalézt vhodnou knihovnu.	– oproti ostatním jazykům obtížnější hledání zdrojů a potřebných knihoven přímo pro problematiku implementace šifer (práce	+ jednodušší hledání zdrojů oproti C#, mnohdy řešení problému nalezeno nejprve v Javě a poté převzato i do C#

		s textem, velkými čísly …)	– oproti Pythonu se případné návody na řešení problémů hledají déle
Jak dlouho trval vývoj	+ nejrychlejší vývoj	– časové nejnáročnější vývoj	+– vývoj v časové náročnosti na 2. místě
Vývojová prostředí	+ snadná práce s vývojovým prostředím + možnost přehledného ladění kódu + zvýrazňování syntaxe + našeptávání kódu	+ snadná práce s vývojovým prostředím + našeptávání kódu + možnost přehledného ladění kódu + zvýrazňování syntaxe	+ snadná práce s vývojovým prostředím – nenašeptává automaticky např. názvy již definovaných proměnných (pouze po napsání 2 teček) – na začátek nepřehledné ladění kódu + zvýrazňování syntaxe
Tvorba GUI	+ přehledná tvorba + není potřeba tvořit GUI pouze psaním kódu, dostupný grafický designer	+ přehledná tvorba – nutnost dávat si pozor na nastavení vlastností jako Scrollbars a MultiLine + není potřeba tvořit GUI pouze psaním kódu	+ přehledná tvorba – nutnost dávat si pozor na velikost okna formuláře + není potřeba tvořit GUI pouze psaním kódu, dostupný grafický designer

8 OPTIMALIZACE A DOBA TRVÁNÍ ŠIFER

V této kapitole jsou konkrétní ukázky možností optimalizace jednotlivých šifer a porovnání výsledků měření doby trvání šifrování a dešifrování při neoptimalizovaném a optimalizovanějším kódu. Některé tipy pro optimalizaci v daných jazycích a možnosti měření času, po jaký daný úsek programu běží, byly rozebrány již kapitole 5.

8.1 Python

V Pythonu je u šifry RSA využito optimalizace pomocí *Generator Expression* a funkce *join()*, a to při převodu textu k zašifrování do binárních čísel (viz Obrázek 83 a Obrázek 84). Generator Expression má podobnou syntaxi jako List Comprehension, rozdílem je pouze ohraničení kulatými závorkami místo hranatých.

```
86          # binarniForma=""
87           #projde se kazdy znak v otevrenem textu
88           #for i in otevrenyTextIn:
89            #   a=ord(i)#prevede se do ascii cisla
90              #  b="{0:b}".format(a)#prevede se do binarniho cisla
91                # bi=self.vlozNuly(b)#vlozi se 0 pred cislo, dokud je mensi nez 11 cislic
92                  #binarniForma+=bi#jednotlive binarni cisla se spoji do jednoho
93
94           #optimalizace text do binarniho cisla
95           binarniForma=self.TextToBin(otevrenyTextIn)
```

Obrázek 83. Optimalizace převodu textu do binárních čísel (Python)

```
130
131     def TextToBin(self,text):
132        #optimalizace prevodu textu na binarni cisla tak, aby kazdy znak byl preveden na b.cislo s delkou 11 pomoci generator
133        binarniText=''.join(self.vlozNuly(format(ord(x), 'b')) for x in text)
134        return binarniText
```

Obrázek 84. Optimalizace převodu textu do binárních čísel 2 (Python)

Dále je zde optimalizován průchod textu v binární podobě po úsecích s délkou 88 a jejich zašifrování, a to s využitím *List Comprehension* (viz Obrázek 85 a Obrázek 86). Jak lze vidět, takovými optimalizacemi dochází i k značnému zredukování řádků kódu.

```
104          # rozdelene=re.findall('.{1,88}',binarniForma) #rozdeli celkove binarni cislo po 88cich
105           #sifrovanyKomplet pro ulozeni celkoveho zasifrovaneho textu
106          # sifrovanyKomplet=[]
107           #projdou se vsechny binarni cisla s delkou 88
108           #for i in rozdelene:
109            #    asc=int(i, 2)#vetsinou po 88cich se prevede do ascii cisla
110              #   st=pow(asc,e,n)#ascii cislo sifrovane podle vzorce OT^e mod n
111                #  sifrovanyKomplet.append(st)#zasifrovana cast textu je pridana do sifrovaneho kompletu
112           #obsah pole sifrovaneho kompletu je spojen pomoci \n a nasledne vypsan uzivateli
113           #sifrovanyKomplet=('\n'.join(map(str, sifrovanyKomplet)))
114
115           #optimalizace prevodu binarnich cisel a jejich zasifrovani
116           sifrovanyKomplet=self.BinToTextSifr(binarniForma,e,n)
```

Obrázek 85. Optimalizace části šifrování pomocí List comprehension (Python)

```
125         def BinToTextSifr(self,binarniForma, e, n):
126             rozdelene=re.findall('.{1,88}',binarniForma)
127             sifrovanyKomplet=[str(pow(int(i,2),e,n)) for i in rozdelene]
128             return '\n'.join(sifrovanyKomplet)
```

Obrázek 86. Optimalizace části šifrování pomocí List comprehension 2 (Python)

U dešifrování RSA je optimalizován for cyklus, který prochází každý prvek listu textu k dešifrování a vrací již dešifrovaný text. Tato optimalizace je provedena opět s využitím Generator Expression, což je k nahlédnutí v kódech v příloze P I.

Šifra OTP a Zubatka jsou optimalizovány taktéž pomocí List Comprehension nebo Generator Expression a funkce *join()*. U Zubatky této možnosti využívají téměř všechny for cykly. Níže jsou uvedeny ukázky kódu s konkrétní aplikací (Obrázek 87 až Obrázek 92).

```
45          # klic=''
46           #for i in range(delkaTextu):
47            #   klic+=str(random.randint(0, 1))
48
49           #optimalizace generovani klice pomoci list comprehensions
50           klic=''.join([str(random.randint(0, 1)) for i in range(delkaTextu)])
```

Obrázek 87. Optimalizace generování klíče OTP (Python)

```
65      # for i in range(int(len(textNaZasifrovaniPole)/len(klicCisla))+1):
66       #    for j in range(len(tabulkaText)):
67        #       if i>=klicCisla[j] and len(textNaZasifrovaniPole)-1>=self.znak:
68         #          tabulkaText[j].append(textNaZasifrovaniPole[self.znak])
69          #         self.znak+=1
70
71
72       #optimalizace doplneni 2.casti tabulky pomoci listComprehensions
73       [self.PomocnaKNaplneniPrvniDruheCastiTabulky(i,j,tabulkaText,textNaZasifrovaniPole) for i in range(int(len(textNaZasifrovaniPole)/len(klicCisla))+1)
74        for j in range(len(tabulkaText)) if i>=klicCisla[j] and len(textNaZasifrovaniPole)-1>=self.znak]
```

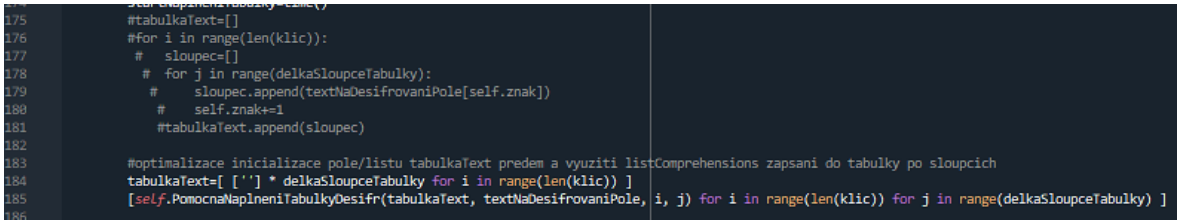
Obrázek 88. Optimalizace naplnění 2. části tabulky – Zubatka (Python)

```
110         def PomocnaKNaplneniPrvniDruheCastiTabulky(self,i,j,tabulkaText,textNaZasifrovaniPole):
111             tabulkaText[j].append(textNaZasifrovaniPole[self.znak])
112             self.znak+=1
```

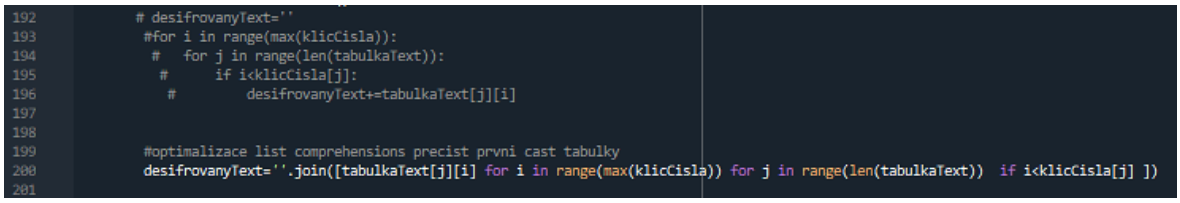
Obrázek 89. Optimalizace naplnění 2. části tabulky 2 – Zubatka (Python)

```
91          # zasifrovanyText=''
92           #for i in range(len(klic)):
93            #   for j in range(len(tabulkaText[0])):
94             #      zasifrovanyText+=tabulkaText[i][j]
95
96
97           #optimalizace list comprehensions - precist a vratit text po sloupcich
98           zasifrovanyText=''.join([tabulkaText[i][j] for i in range(len(klic)) for j in range(len(tabulkaText[0]))]  )
```

Obrázek 90. Optimalizace šifrování – Zubatka – čtení textu po sloupcích (Python)

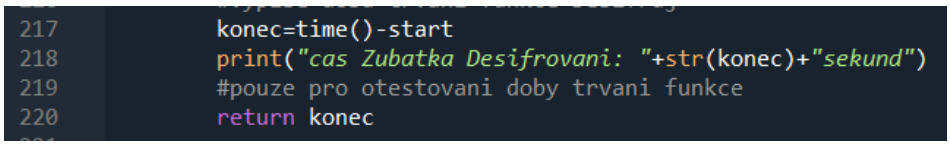


Obrázek 91. Optimalizace dešifrování – Zubatka – doplnění textu do tabulky po sloupcích (Python)

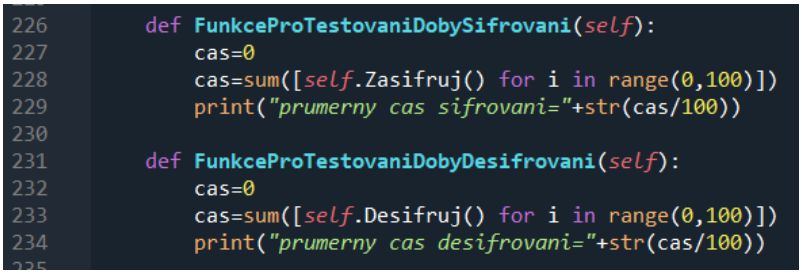


Obrázek 92. Optimalizace dešifrování – Zubatka – čtení 1. části tabulky (Python)

Měření doby trvání funkcí v Pythonu je prováděno pomocí knihovny *time* a její funkce *time()*. Na začátek souborů je tedy vložen import knihovny: *from time import time*. Na začátku kódu, u kterého je žádoucí změřit čas běhu, je zavedena proměnná *start*, do které je uložena návratová hodnota z funkce *time()*. Jedná se o číslo, které vyjadřuje čas v sekundách od epochy v UTC. Na konci této části kódu je definována proměnná *konec*, do které se ukládá opět hodnota z funkce *time()*, od které je odčítána hodnota proměnné *start* (viz Obrázek 93). Tato hodnota konce je za účelem měření průměrné doby trvání šifrování a dešifrování vrácena. Dále jsou definovány funkce, ve kterých probíhá cyklus 100 šifrování nebo dešifrování a vrací průměrnou hodnotu doby trvání prováděných funkcí (viz Obrázek 94).



Obrázek 93. Ukázka definice proměnné konec a jejího následného vrácení na konci funkce (Python)



Obrázek 94. Ukázka zavedených funkcí pro výpočet doby trvání (Python)

Pro porovnání optimalizovaného versus neoptimalizovaného kódu bylo prováděno měření u jednotlivých šifer s následujícími vstupy:

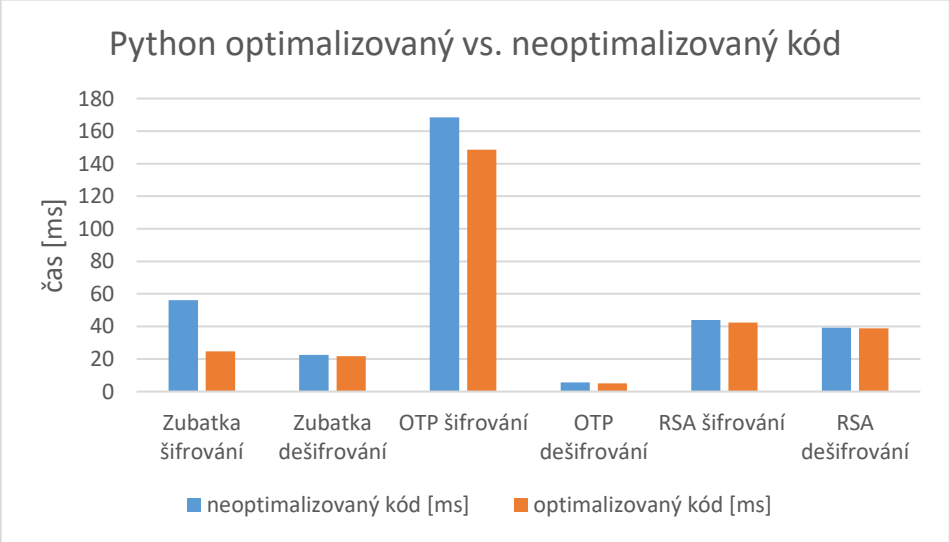
- RSA – vstupní text k šifrování o délce 5000 náhodných znaků
- OTP – vstupní text o délce 1000 znaků
- Zubatka – vstupní text o délce 10000 znaků s klíčem *totojeklicprotestovanicasu*

Výsledky měření jsou převedeny do milisekund a znázorněny pomocí tabulky a grafu níže (Tabulka 25, Obrázek 95).

Tabulka 25. Výsledky doby trvání – optimalizovaný a neoptimalizovaný kód (Python)

	neoptimalizovaný kód [ms]	**optimalizovaný kód [ms]**
Zubatka šifrování	56,120	24,679
Zubatka dešifrování	22,527	21,754
OTP šifrování	168,468	148,521
OTP dešifrování	5,656	5,060
RSA šifrování	43,953	42,341
RSA dešifrování	39,219	38,775

Z výsledků lze vidět, že největšího zrychlení optimalizací dosáhla funkce šifrování u šifry Zubatka. Z dat je viditelné, že vždy došlo k alespoň malému urychlení programu.

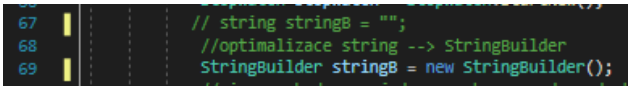


Obrázek 95. Graf – Python optimalizovaný vs. neoptimalizovaný kód

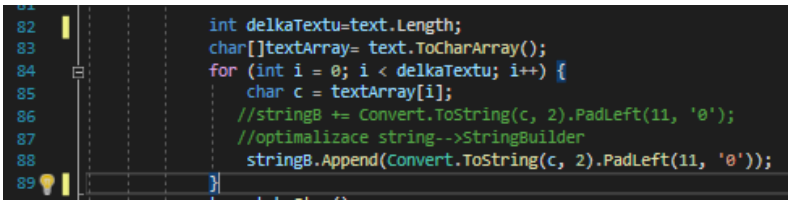
8.2 C#

V C# u šifry RSA je nejdůležitější optimalizací zrychlení ověřování prvočísel, jelikož se zde program nejdéle zdržoval. Tato optimalizace je provedena ve dvou verzích. Jednou z nich je vytvoření nové třídy a vložení kódu s Rabin – Miller testem pro ověření prvočísel datového typu BigInteger. Zde se objevuje i další možnost, a to použití rovnou metody datového typu BigInteger (tato možnost zde je, jelikož je použit balíček BouncyCastle). Ukázka použití daných funkcí je znázorněna výše v kapitole 7.1.2 na Obrázek 48. Dále je u šifer využito možnosti optimalizace v podobě použití datového typu StringBuilder namísto string tam, kde je potřeba s řetězcem více pracovat (např. přidávat do něj text). Další optimalizací je nahrazení foreach cyklu cyklem for nebo definování přibližné velikosti Listu (použito u šifry Zubatka).

StringBuilder použit při převodu textu do řetězce binárních čísel u šifry RSA je znázorněn na Obrázek 96 a Obrázek 97.

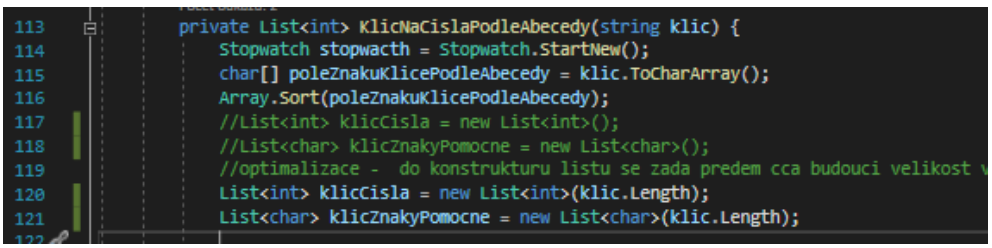


Obrázek 96. Optimalizace převodu textu do binárního řetězce RSA (C#)



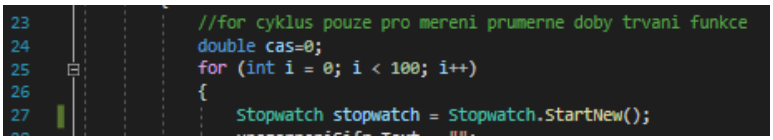
Obrázek 97. Optimalizace převodu textu do binárního řetězce RSA část 2 (C#)

Další příklad využití StringBuilder místo stringu je znázorněn v kapitole 7.3.2 při převodu řetězce binárních čísel do řetězce znaků (konkrétně na Obrázek 79) nebo u šifry Zubatka při čtení textu z tabulky. Možnost optimalizace definováním přibližné velikosti Listu je využita při šifrování, dešifrování a převodu klíče na čísla podle abecedy u šifry Zubatka. Názorný příklad lze vidět na Obrázek 98.

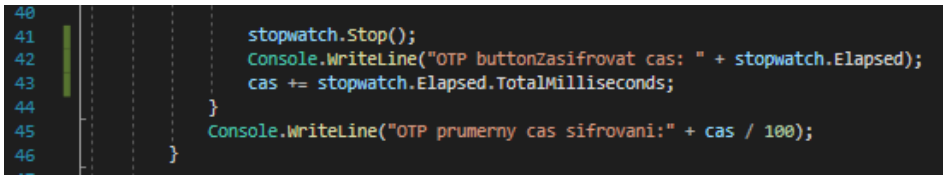


Obrázek 98. Optimalizace převodu klíče na čísla podle posloupnosti abecedy (C#)

V C# probíhá měření doby trvání programu pomocí třídy *Stopwatch*. Na začátku úseku kódu, který je měřen, je definována proměnná typu Stopwatch a stopky jsou zároveň zapnuty, a to pomocí příkazu *Stopwatch.StartNew()*. Na konci tohoto úseku jsou stopky vypnuty funkcí *Stop()*. Pokud je třeba stejné stopky znovu zapnout, musí být jejich hodnota před opětovným spuštěním resetována, jinak by se pouze pokračovalo v přičítání k času, který byl stopnut dříve. Kvůli měření celé funkce šifrování a dešifrování je zavedena proměnná *cas* a for cyklus, ve kterém šifrování nebo dešifrování probíhá (viz Obrázek 99 a Obrázek 100). Do proměnné *cas* je přičítána doba trvání v milisekundách a po proběhnutí sta cyklů je vypsán průměrný čas, po který program běží.



Obrázek 99. Ukázka měření doby trvání funkcí (C#)



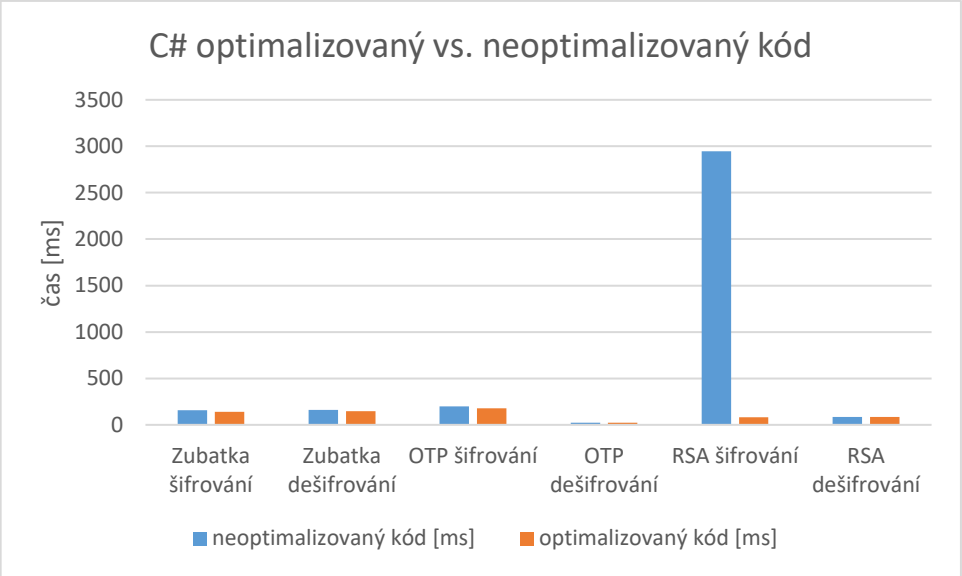
Obrázek 100. Ukázka měření doby trvání funkcí část 2. (C#)

Za účelem měření času bylo použito vstupních hodnot jako v případě jazyka Python. Tabulka naměřených průměrných časů v milisekundách je k nahlédnutí níže (Tabulka 26).

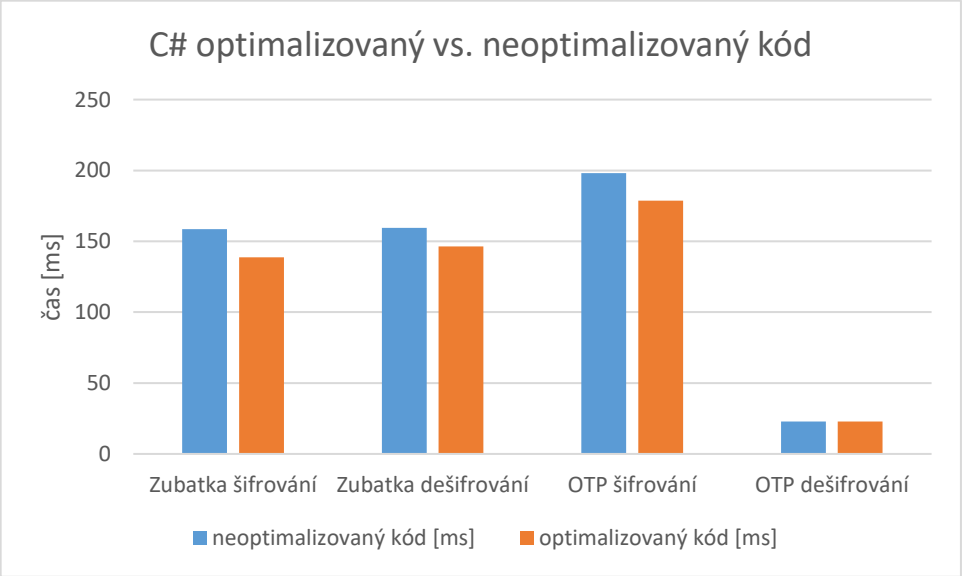
Tabulka 26. Výsledky doby trvání – optimalizovaný vs. neoptimalizovaný kód (C#)

	neoptimalizovaný kód [ms]	**optimalizovaný kód [ms]**
Zubatka šifrování	158,676	138,677
Zubatka dešifrování	159,440	146,470
OTP šifrování	198,005	178,609
OTP dešifrování	22,954	22,851
RSA šifrování	2946,042	80,596
RSA dešifrování	86,627	85,576

Z tabulky lze vyčíst, že nejviditelnějšího zrychlení dosáhla funkce šifrování RSA šifry. Z tabulky taktéž vyplývá, že každá funkce dosáhla po optimalizaci zrychlení. Pro ještě lepší názornost jsou naměřená data znázorněna v grafech níže (Obrázek 101 a Obrázek 102). Jelikož je hodnota neoptimalizovaného RSA šifrování poměrně vysoká oproti hodnotám ostatním, jsou z důvodu lepší čitelnosti rozdílu časové náročnosti vytvořeny dva grafy.



Obrázek 101. Graf – C# optimalizovaný vs. neoptimalizovaný kód (1)



Obrázek 102. Graf – C# optimalizovaný vs. neoptimalizovaný kód (2)

8.3 Java

V Javě, podobně jako v C#, je nejdůležitější a taky nejsložitější provést optimalizaci části kódu, ve které probíhá ověřování prvočísel u šifry RSA. V projektu je vytvořen další soubor

MillerRabin.java, ve kterém je vložen program ověřující prvočíslo. Oproti C# zde použitý datový typ BigInteger přímo nenabízí metodu, která by prvočíslo ověřovala. Dále jsou v šifrách využity možnosti optimalizace jako použití datového typu StringBuilder namísto String, definování velikosti Listu nebo optimalizace for cyklu zavedením lokálních proměnných, bez kterých by byly v cyklu neustále volány různé funkce (například pro zjištění velikosti procházeného pole). Níže jsou uvedeny ukázky konkrétního použití v šifrách.

Možnost použití StringBuideru namísto Stringu je znázorněna níže (Obrázek 103).

```
65      //generovani klice v podobe random binarnich cisel o delce jako vstupni data v binarni podobe
66      private String vygenerujBinKlic(int delkaVstupu){
67          long start=System.nanoTime();
68          //String klic ="";
69          //optimalizace String --> StringBuilder
70          StringBuilder klic=new StringBuilder();
71          Random random = new Random();
72
73          for(int i=0;i<delkaVstupu;i++){
74            // klic+=random.nextInt(2);
75             //optimalizace String --> StringBuilder
76             klic.append(random.nextInt(2));
77          }
78         long konec = System.nanoTime() - start;
79         System.out.print("OTP vygenerujBinKlic String/StringBuilder cas:"+konec+"\n");
80          //return klic;
81          //kvuli StringBuilder:
82          return klic.toString();
83      }
```

Obrázek 103. Optimalizace generování klíče OTP (Java)

Ukázka optimalizace for cyklu je viditelná na Obrázek 104.

```
113         /*for(int sloupecc=0;sloupecc<klicSifr.length();sloupecc++){
114             while(tabulkaText.get(sloupecc).size()<maximalniDelkaSloupce.get().size()){
115                 tabulkaText.get(sloupecc).add(abeceda.charAt(rd.nextInt(abeceda.length())));
116             }
117         }*/
118
119         //optimalizace for cyklu
120         int klicCislaSize=klicCisla.size();
121         int maximalniDelkaSloupceSize=maximalniDelkaSloupce.get().size();
122
123         for(int sloupecc=0;sloupecc<klicCislaSize;sloupecc++){
124             while(tabulkaText.get(sloupecc).size()<maximalniDelkaSloupceSize){
125                 tabulkaText.get(sloupecc).add(abeceda.charAt(rd.nextInt(abeceda.length())));
126             }
127         }
```

Obrázek 104. Optimalizace doplnění tabulky náhodnými písmeny – šifrování Zubatka (Java)

Optimalizace pomocí definování velikosti Listu je znázorněna na příkladu převodu klíče na čísla podle posloupnosti znaků v abecedě (viz Obrázek 105).

```
158     private List<Integer> KlicNaCislaPodleAbecedy(String klic){
159         long start = System.nanoTime();
160         char poleZnakuKliceSerazene[]=klic.toCharArray();
161         Arrays.sort(poleZnakuKliceSerazene);
162         //List<Integer> klicCisla = new ArrayList<>();
163         //List<Character> klicZnakyPomocne=new ArrayList<>();
164         //optimalizace - urcena priblizna velikost listu
165         List<Integer> klicCisla = new ArrayList<>(klic.length());
166         List<Character> klicZnakyPomocne=new ArrayList<>(klic.length());
167         char[]poleKlice=klic.toCharArray();
```

Obrázek 105. Optimalizace převodu klíče na čísla dle posloupnosti abecedy (Java)

Doba trvání úseků kódu v Javě je měřena pomocí funkce *System.nanoTime()*, která vrací hodnotu času v nanosekundách. Na začátku úseků kódů, které je zapotřebí změřit, je definována proměnná *start* datového typu long, která obsahuje návratovou hodnotu právě funkce *System.nanoTime()*. Na konci úseku kódu je definována proměnná *konec*, ve kterém je obsažena hodnota proměnné *start* odečtena od aktuálního času v nanosekundách. Pro změření průměrné doby šifrování a dešifrování je v projektech ve složkách PrimaryController.java ve funkcích šifrování a dešifrování definována proměnná *cas* a kód šifrování a dešifrování je vložen do for cyklu. V každém for cyklu je tedy stanovena hodnota nanosekund, která je přičtena k celkovému času. Ten je nakonec podělen 100 a odpovídá průměrné hodnotě doby trvání (viz Obrázek 106 a Obrázek 107).

```
42      @FXML
        private void zasifrovat()throws IOException{
44          //promenna cas a for cyklus pouze z duvodu mereni prumerne doby trvani cyklu
45          long cas=0;
46          for(int i = 0; i<100;i++){
47          long start =System.nanoTime();
```

Obrázek 106. Ukázka měření doby trvání funkcí (Java)

```
65          long konec = System.nanoTime()-start;
66          System.out.print("RSA zasifrovat cas: "+konec +"\n");
67          cas+=konec;
68          }
69          System.out.print("RSA prumerna doba sifrovani: "+cas/100 +" nanosekund");
70      }
```

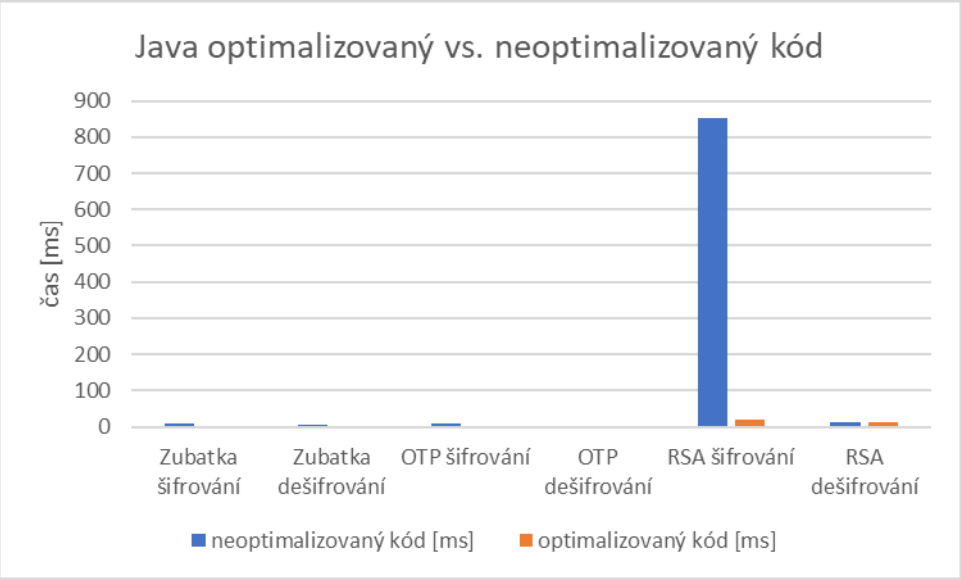
Obrázek 107. Ukázka měření doby trvání funkcí část 2 (Java)

Při měření času běhu optimalizovaných a neoptimalizovaných programů bylo u jednotlivých šifer využito vstupních dat totožných jako v Pythonu. Výsledky měření převedeny do milisekund jsou znázorněny pomocí tabulky a grafů níže (Tabulka 27, Obrázek 108 a Obrázek 109). Protože je hodnota neoptimalizovaného RSA šifrování opět poměrně vysoká oproti ostatním hodnotám, jsou z důvodu lepší čitelnosti rozdílu časové náročnosti vytvořeny dva grafy.

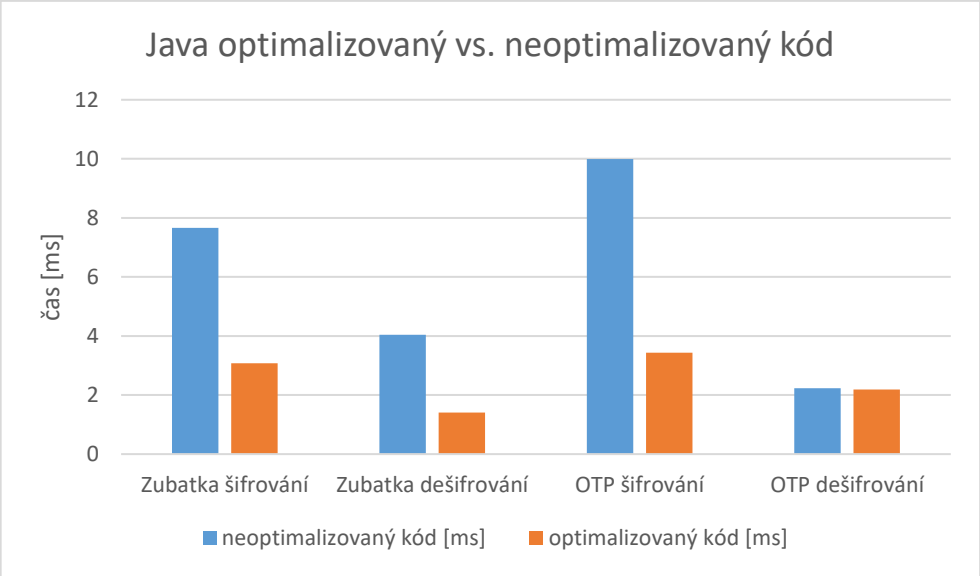
Tabulka 27. Výsledky doby trvání – optimalizovaný vs. neoptimalizovaný kód (Java)

	neoptimalizovaný kód [ms]	optimalizovaný kód [ms]
Zubatka šifrování	7,657	3,073
Zubatka dešifrování	4,041	1,410
OTP šifrování	9,997	3,436
OTP dešifrování	2,236	2,191
RSA šifrování	852,589	20,534
RSA dešifrování	12,890	12,710

Při optimalizaci došlo vždy alespoň k malému zlepšení výkonu programu. Největší změnu lze pozorovat u šifry RSA, a to konkrétně při šifrování.



Obrázek 108. Graf – Java optimalizovaný vs. neoptimalizovaný kód (1)



Obrázek 109. Graf – Java optimalizovaný vs. neoptimalizovaný kód (2)

9 POPIS WEBOVÉ STRÁNKY

Webová stránka je vytvořena jako přehled o implementaci šifer v jednotlivých jazycích. Je vytvořena v prostředí Visual Studia jako webová aplikace ASP.NET Core. Dochází k využití MVC architektury. Projekt je strukturován do jednotlivých oblastí. Každá oblast obsahuje kontroler a jednotlivá zobrazení. Ukázku kódu kontroleru a zobrazení lze vidět na Obrázek 110 a Obrázek 111.

```
using Microsoft.AspNetCore.Mvc;
using System;
using System.Collections.Generic;
using System.Linq;
using System.Threading.Tasks;


namespace bakalarkaWeb.Areas.RSAsifra.Controllers
{
    [Area("RSAsifra")]
    Počet odkazů: 0
    public class RSAsifraController : Controller
    {
        Počet odkazů: 0 | 0 žádostí | 0 výjimky
        public IActionResult Index()
        {
            return View();
        }

        Počet odkazů: 0 | 0 žádostí | 0 výjimky
        public IActionResult RSA_C()
        {

            return View();
        }

        Počet odkazů: 0 | 0 žádostí | 0 výjimky
        public IActionResult RSA_Java()
        {

            return View();
        }
        Počet odkazů: 0 | 0 žádostí | 0 výjimky
        public IActionResult RSA_Python()
        {

            return View();
        }

    }
}
```

Obrázek 110. Ukázka kontroleru u oblasti RSA

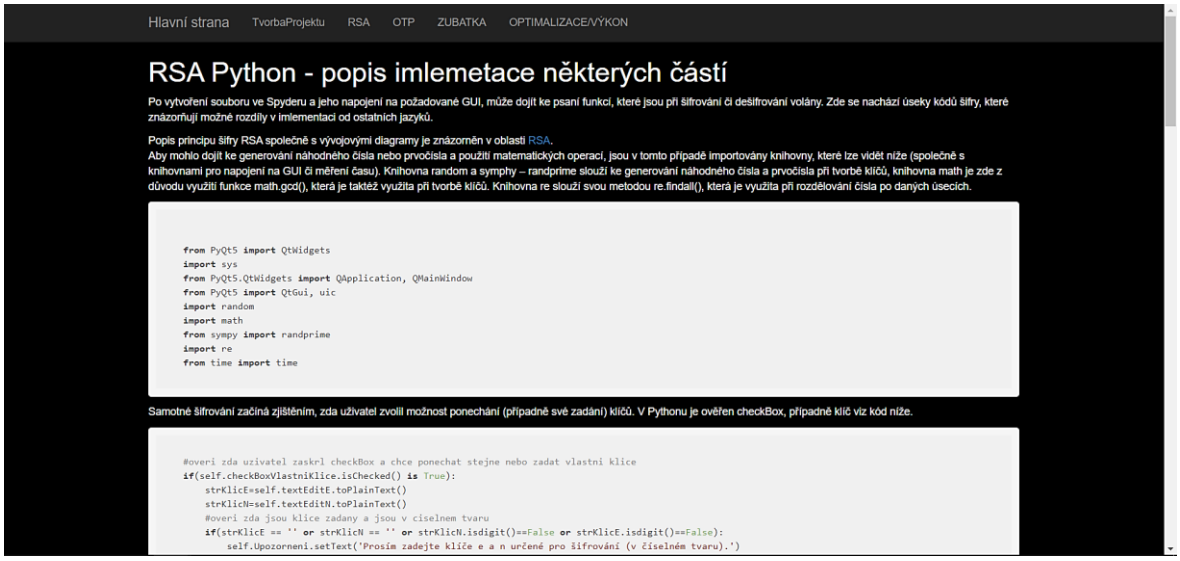
```
@{
    ViewData["Title"] = "RSA_Python";
}
<h1>RSA Python - popis imlemetace některých částí</h1>
<p>Po vytvoření souboru ve Spyderu a jeho napojení na požadované GUI, může dojít ke psaní funkcí, které jsou při šifrování či dešifrování volány.
    Zde se nachází úseky kódů šifry, které znázorňují možné rozdíly v imlementaci od ostatních jazyků.</p>
<div>
    <p>
        Popis principu šifry RSA společně s vývojovými diagramy je znázorněn v oblasti  <a asp-area="RSAsifra" asp-controller="RSAsifra" asp-action="Index">RSA</a>.
        <br>
        Aby mohlo dojít ke generování náhodného čísla nebo prvočísla a použití matematických operací, jsou v tomto případě importovány knihovny,
        které lze vidět níže (společně s knihovnami pro napojení na GUI či měření času).
        Knihovna random a sympy - randprime slouží ke generování náhodného čísla a prvočísla při tvorbě klíčů,
        knihovna math je zde z důvodu využití funkce math.gcd(), která je taktéž využita při tvorbě klíčů.
        Knihovna re slouží svou metodou re.findall(), která je využita při rozdělování čísla po daných úsecích.
    </p>
    <pre><code>

    from PyQt5 import QtWidgets
    import sys
    from PyQt5.QtWidgets import QApplication, QMainWindow
    from PyQt5 import QtGui, uic
    import random
    import math
    from sympy import randprime
    import re
    from time import time

    </code></pre>
    <p>
        Samotné šifrování začíná zjištěním, zda uživatel zvolil možnost ponechání (případně své zadání) klíčů. V Pythonu je ověřen checkBox, případně klíč viz kód níže.
    </p>
```

Obrázek 111. Ukázka kódu zobrazení RSA_Python

Na hlavní stránce (viz Obrázek 112) má uživatel vypsána vývojová prostředí, která jsou použita pro implementaci šifer v jednotlivých jazycích. Přes prvek navbar se dále může dostat do oblasti TvorbaProjektu, kde je popsáno vytvoření GUI a napojení na soubory, ve kterých je psána funkcionalita šifrování a dešifrování. V oblasti RSA, OTP či ZUBATKA se nabízí

čtyři možnosti. Prvek dropdown menu nabízí u každé šifry přesměrování na stránku s názvem šifry, kde je dostupný princip implementace šifer společně s vývojovými diagramy. Prvky dropdown menu s názvem šifry a jazyka přesměrují uživatele na stránku, kde jsou popsány některé části kódu a možné rozdíly v jednotlivých jazycích (viz Obrázek 113). Pro zobrazování úseků kódu, které lze jednoduše zkopírovat, je na webové stránce využito tagů <pre> a <code>. Pro zvýraznění syntaxe je využit hightlight.js. Po kliknutí na poslední prvek navbaru – OPTIMALIZACE/VÝKON se uživateli zobrazí dropdown menu s možností přechodu na informace o možnostech optimalizace a grafy porovnání optimalizovaného a neoptimalizovaného kódu v daných jazycích.

Obrázek 112. Ukázka hlavní stránky webové aplikace



Obrázek 113. Ukázka zobrazení RSA – RSA_Python

ZÁVĚR

Cílem bakalářské práce bylo poskytnout „první pomoc“ při výběru programovacího jazyka v předmětu Kryptologie. Tato volba na začátku semestru se pro mnohé studenty může stát poněkud osudnou a při nesprávné volbě jazyka si mohou způsobit zbytečné komplikace v průběhu psaní šifer. V teoretické části je student seznámen se zástupci šifer z oblasti klasické i moderní kryptografie a principy průběhů šifrování. Pro lepší pochopení principů jednotlivých šifer, jsou zde použity tabulky a obrázky s jejich objasněním. Dále jsou zde rozebrány možnosti optimalizace kódu v daných jazycích, problematika časové náročnosti programů a možnosti měření doby trvání jednotlivých úseků kódu.

V praktické části je popsán postup práce s vývojovými prostředími zvolených u každého z jazyků (Python, C#, Java), a to jak při tvorbě grafického rozhraní, tak při psaní již samotné funkcionality šifrování a dešifrování. V oblasti popisu tvorby GUI jsou taktéž popsány některé výhody či nevýhody u jednotlivých jazyků. Nejjednodušeji se GUI vytváří v prostředí Qt Designer (Python), kde nemusí být kladena tak velká pozornost na nastavování funkčních vlastností jako je možnost scrollování nebo zobrazení více řádků u textových polí (což je například ve Visual Studiu nutné). Je zde taky velmi intuitivní popisování jednotlivých prvků.

Dále jsou implementovány šifry RSA, Zubatka a One-time Pad. Je zde popsána jejich implementace a možné rozdíly v průběhu psaní kódu. Z hlediska náročnosti práce vychází opět nejlépe jazyk Python. Má dostupné velké množství knihovních funkcí, které při šifrování velmi usnadňují práci a také jednodušší syntaxi. V jazyku C# je nejobtížnější najít vhodný datový typ nebo napsat funkce pro ověřování prvočísel či generování čísel v určitém rozsahu. Java je na tom z hlediska náročnosti práce podobně jako jazyk C#.

Dále jsou zde znázorněny možnosti optimalizace šifer ve vybraných jazycích na konkrétních příkladech. Python nabízí optimalizace jako využití List Comprehension nebo GeneratorExpression, což umožňuje i značné redukování řádků kódu. V případě jazyků C# a Java se nabízí optimalizace ve využití datového typu StringBuilder namísto stringu, definování alespoň přibližných velikostí listů či definice proměnných před použitím ve for cyklech. Naměřené hodnoty doby trvání šifrování a dešifrování v případě neoptimalizovaného kódu jsou větší než v případě aplikace optimalizací. To vše je znázorněno i pomocí tabulek a grafů. Z hlediska výkonnosti při spuštění šifrování a dešifrování vychází nejlépe jazyk Java, následuje Python a jako poslední C#.

Nakonec byla jako součást bakalářské práce vytvořena webová stránka pomocí .NET Core ve Visual Studiu. Webová stránka shrnuje všechny poznatky od tvorby GUI, přes popis implementace některých částí kódu, ukázku všech vývojových diagramů až po ukázky optimalizace konkrétních částí kódu. Na hlavní stránce jsou stručně popsány výhody a nevýhody použití vybraných jazyků.

Dalším rozšířením této práce by mohla být implementace šifer a popis kladů a záporů i v dalších jazycích, například ve Wolfram Mathematice.

Tato bakalářská práce bude sloužit jako přehled o možnostech implementace šifer v jazycích Python, Java a C#. Poskytuje studentům souhrn zjištěných výhod a nevýhod při implementaci šifer, tudíž každý student se bude schopen lépe rozhodnout, který jazyk pro něj bude ten nejvýhodnější. Práce bude sloužit jako výukový materiál do začátku semestru předmětu Kryptologie.

SEZNAM POUŽITÉ LITERATURY

[1] Úvod do kryptologie. *Mendelova univerzita v Brně [CZ]* [online]. [cit. 2021-01-27]. Dostupné z: https://is.mendelu.cz/eknihovna/opory/zobraz_cast.pl?cast=7021

[2] PIPER, F. C. a Sean MURPHY. *Kryptografie*. Praha: Dokořán, 2006. Průvodce pro každého. ISBN 80-736-3074-5.

[3] SINGH, Simon. *Kniha kódů a šifer: Utajování od starého Egypta po kvantovou kryptografii*. Praha: Dokořán, 2003. Aliter (Argo: Dokořán): Dokořán). ISBN 80-86569-18-7.

[4] Cryptography. *TechTarget SearchSecurity* [online]. 2018 [cit. 2021-01-27]. Dostupné z: https://searchsecurity.techtarget.com/definition/cryptography

[5] Difference Between Substitution Technique and Transposition Technique. *TechDifferences* [online]. 2018 [cit. 2021-01-28]. Dostupné z: https://techdifferences.com/difference-between-substitution-technique-and-transposition-technique.html

[6] LURINGEN. Skytale. In: *WIKIMEDIA COMMONS* [online]. 20 February 2007 [cit. 2021-01-28]. Dostupné z: https://commons.wikimedia.org/wiki/File:Skytale.png

[7] Caesarova šifra. *Algoritmy.net* [online]. [cit. 2021-01-28]. Dostupné z: https://www.algoritmy.net/article/34/Caesarova-sifra

[8] RODRIGUEZ-CLARK, Daniel, c2013-2019. The ciphertext alphabet for the Atbash Cipher. In: *CRYPTO CORNER* [online]. [cit. 2021-01-28]. Dostupné z: https://crypto.interactive-maths.com/atbash-cipher.html

[9] RODRIGUEZ-CLARK, Daniel, c2013-2019. Atbash Cipher. *CRYPTO CORNER* [online]. [cit. 2021-01-28]. Dostupné z: https://crypto.interactive-maths.com/atbash-cipher.html

[10] Keyword Cipher, *GeeksforGeeks* [online]. Uttar Pradesh [cit. 2021-01-28]. Dostupné z: https://www.geeksforgeeks.org/keyword-cipher/

[11] HODAN, George, 2020. Morseovka: Morseova abeceda. In: *PublicDomainPictures.net* [online]. England and Wales: Bobek [cit. 2021-01-31]. Dostupné z: https://www.publicdomainpictures.net/pictures/40000/velka/morse.jpg

[12] ESHAM, Benjamin D., A Vigenère square. In: *WIKIPEDIE: Otevřená encyklopedie* [online]. [cit. 2021-01-31]. Dostupné z:

https://upload.wikimedia.org/wikipedia/commons/thumb/2/25/Vigen%C3%A8re_square.svg/1024px-Vigen%C3%A8re_square.svg.png

[13] BERLOQUIN, Pierre, 2011. *Skryté kódy a velkolepé projekty: tajné jazyky od starověku po současnost*. Praha: Knižní klub, s. 193-194. Universum (Knižní klub). ISBN 9788024228471.

[14] LYONS, James, 2012. Playfair Cipher, *Practical Cryptography* [online]. [cit. 2021-02-01]. Dostupné z: http://practicalcryptography.com/ciphers/polygraphic-substitution-category/playfair/

[15] RAMSWARUP_KULHARY. Bifid Cipher in Cryptography. *GeeksforGeeks* [online]. Uttar Pradesh, 06 Nov, 2019 [cit. 2021-02-01]. Dostupné z: https://www.geeksforgeeks.org/bifid-cipher-in-cryptography/

[16] LYONS, James, 2012. Bifid Cipher, *Practical Cryptography* [online]. [cit. 2021-02-01]. Dostupné z: http://practicalcryptography.com/ciphers/bifid-cipher/

[17] LYONS, James, 2012. Hill Cipher. *Practical Cryptography* [online]. [cit. 2021-02-01]. Dostupné z: http://practicalcryptography.com/ciphers/hill-cipher/

[18] JANEČEK, Jiří, 1994. *Odhalená tajemství šifrovacích klíčů minulosti: ruční šifry*. Praha: Naše vojsko. Mozaika (Naše vojsko). ISBN 80-206-0462-6.

[19] SIMMONS, Gustavus J. Cryptology: Cipher systems. *Encyclopædia Britannica* [online]. c2021 [cit. 2021-02-02]. Dostupné z: https://www.britannica.com/topic/cryptology/Cryptography#ref233468

[20] LYONS, James, 2012. ADFGX Cipher. *Practical Cryptography* [online]. [cit. 2021-02-02]. Dostupné z: http://practicalcryptography.com/ciphers/adfgx-cipher/

[21] POLÁK, Ing. Karel. Bezpečná komunikace uživatelů. *Elektrorevue* [online]. Brno, 2.4.2002 [cit. 2021-02-02]. Dostupné z: http://www.elektrorevue.cz/clanky/02021/index.html

[22] What Is Symmetric Key Cryptography? *BINANCE ACADEMY* [online]. c2021 [cit. 2021-02-02]. Dostupné z: https://academy.binance.com/en/articles/what-is-symmetric-key-cryptography

[23] PAAR, Christof a Jan PELZL. *Understanding Cryptography: A Textbook for Students and Practitioners*. Berlin, Heidelberg: Springer Berlin Heidelberg, 2010. ISBN 978-3-642-44649-8.

[24] TOOB. Digital Signature diagram. In: *WIKIPEDIE: Otevřená Encyklopedie* [online]. 29. června 2009 [cit. 2021-02-03]. Dostupné z: https://cs.wikipedia.org/wiki/Elektronick%C3%BD_podpis#/media/Soubor:Digital_Signature_diagram_cs.svg

[25] LAKE, JOSH, 2019. What are digital signatures and how do they work? *Comparitech* [online]. United Kingdom: Comparitech Limited [cit. 2021-02-03]. Dostupné z: https://www.comparitech.com/blog/information-security/digital-signatures/

[26] ASTORI, Anna. Optimizing Your Python Code: Essential tips for working with large amounts of data in Python. *Towards: data science* [online]. May 29, 2020 [cit. 2021-02-04]. Dostupné z: https://towardsdatascience.com/optimizing-your-python-code-156d4b8f4a29

[27] ANSARI, Afzal. Optimization Tips for Python Code. *GeeksforGeeks* [online]. Uttar Pradesh, 19 Nov, 2020 [cit. 2021-02-04]. Dostupné z: https://www.geeksforgeeks.org/optimization-tips-python-code/

[28] SHIRAZI, Jack. *Java – vylaďování výkonu: [účinné a efektivní strategie vylaďování]*. Praha: Grada, 2003. ISBN 80-247-0752-7.

[29] JANSSEN, Thorben. 11 Simple Java Performance Tuning Tips. *Stackify* [online]. Mission, KS, © 2020, AUGUST 13, 2019 [cit. 2021-02-05]. Dostupné z: https://stackify.com/java-performance-tuning/

[30] MOLYNEAUX, Ian. *The Art of Application Performance Testing.* Sebastopol, CA: O'Reilly, 2009. ISBN 978-0-596-52066-3.

[31] SOURAV, Kayal. 5 Tips to improve performance of C# code. *C# Corner* [online]. c2021, Jun 03, 2016 [cit. 2021-02-16]. Dostupné z: https://www.c-sharpcorner.com/UploadFile/dacca2/5-tips-to-improve-performance-of-C-Sharp-code/

[32] ROBERT C., Martin. *Čistý kód: návrhové vzory, refaktorování, testování a další techniky agilního programování*. Brno: Computer Press, 2009. ISBN 978-80-251-2285-3.

[33] NICHOLSON, Cody. Complexity Theory for Algorithms: How we measure the speed of our algorithms. *Medium: BetterProgramming* [online]. Apr 14, 2020 [cit. 2021-02-22]. Dostupné z: https://medium.com/better-programming/complexity-theory-for-algorithms-fabd5691260d

[34] *2009_Sumperk_p10: Časová složitost algoritmů* [online]. Šumperk, 2009 [cit. 2021-02-23]. Dostupné z: https://cw.fel.cvut.cz/old/_media/courses/y36alg/2009_sumperk_p10.pdf

[35] CMGLEE. Comparison computational complexity. In: *WIKIPEDIA: The Free Encyclopedia* [online]. San Francisco (CA): Wikimedia Foundation, 23 June 2017 [cit. 2021-02-23]. Dostupné z: https://en.wikipedia.org/wiki/Time_complexity#/media/File:Comparison_computational_complexity.svg

[36] Timeit — Measure execution time of small code snippets. *Python* [online]. Python Software Foun-dation, c2001-2021 [cit. 2021-02-23]. Dostupné z: https://docs.python.org/3/library/timeit.html

[37] Stopwatch Class. *Microsoft* [online]. Microsoft, c 2021 [cit. 2021-02-23]. Dostupné z: https://docs.microsoft.com/en-us/dotnet/api/system.diagnostics.stopwatch?redirectedfrom=MSDN&view=net-5.0

[38] Class System. *JavaTM 2 Platform Standard Ed. 5.0* [online]. Santa Clara, California 95054, U.S.A: Oracle, c2004, 2010 [cit. 2021-02-23]. Dostupné z: https://docs.oracle.com/javase/1.5.0/docs/api/java/lang/System.html#nanoTime()

[39] Class Stopwatch [online]. c2010-2014 [cit. 2021-02-23]. Dostupné z: https://guava.dev/releases/18.0/api/docs/com/google/common/base/Stopwatch.html

SEZNAM POUŽITÝCH SYMBOLŮ A ZKRATEK

3DES	Triple Data Encryption Standard
AES	Advanced Encryption Standart
Apod.	A podobně
BP	Bakalářská práce
DES	Data Encryption Standard
GUI	Grafické uživatelské rozhraní
ID	IDentification
IDE	Integrated Development Environment
JIT	Just in Time
JVM	Java Virtual Machine
MVC	Model-view-controller
Např.	Například
OT	Otevřený text
OTP	One-time Pad
p.	Pan
RSA	Rivest–Shamir–Adleman
ŠT	Šifrovaný text
UI	Uživatelské rozhraní
UTC	Coordinated Universal Time
VS.	Versus

SEZNAM OBRÁZKŮ

SEZNAM TABULEK

SEZNAM PŘÍLOH

PŘÍLOHA P I: OBSAH CD

Struktura obsahu CD, které je přiloženo:

- BP v elektronické podobě
- Adresář *VyvojoveDiagramy* obsahuje obrázky všech vývojových diagramů šifrování a dešifrování jednotlivých šifrovacích algoritmů
- Adresáře *PythonSifry, JavaSifry a CsharpSifry* obsahují soubory se zdrojovými kódy k jednotlivým jazykům a šifrám
- Adresář *WebovaStranka* obsahuje soubory ke spuštění webové aplikace ve Visual Studiu